Język C++, studia językowe

Michał Wilde

1 Struktura programu

Poniższy przykład pokazuje najprostszy program w języku C. Efektem działania programu będzie napis "Witaj świecie!" wysłany do standardowego urządzenia wyjściowego.

```
// program drukujący napis "Witaj świecie!"

#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Witaj świecie!\n");
  return 0;
}
```

Pierwsza linia programu to komentarz. Dwa ukośniki oznaczają, że tekst następujący po nich aż do końca linii jest ignorowany. Jeśli jest potrzebny komentarz w środku linii to można go zacząć ukośnikiem z gwiazdką (`/*`) i zakończyć gwiazdką z ukośnikiem (`*/`).

W trzeciej linii (uwzględniając linie puste) mamy dyrektywę kompilatora, która nakazuje kompilatorowi wczytanie standardowego pliku nagłówkowego ze standardowymi operacji wejścia/wyjścia. Pliki nagłówkowe są plikami tekstowymi. Bardzo wiele takich plików jest dostarczanych z kompilatorem. Oprócz tego użytkownik może tworzyć własne pliki nagłówkowe. Pliki nagłówkowe są potrzebne kompilatorowi aby wiedział jakie funkcje są dostępne i jakie mają parametry. Sens pisania własnych plików nagłówkowych jest dopiero wtedy gdy nasz program podzielimy na moduły.

Właściwe wykonanie programu rozpoczyna się od wywołania funkcji main. Każda funkcja w języku C składa się z dwóch części: nagłówka funkcji i ciała funkcji. Nagłówek funkcji składa się z deklaracji typu wyniku (tu: `int` czyli liczba całkowita), nazwy funkcji (tu: `main`) i listy parametrów formalnych (tu dwa parametry: `argc` typu całkowitego i `argv`, który jest tablicą wskaźników na łańcuchy). Nagłówek tej wyjątkowej funkcji main jest narzucony. Wynik to kod błędu. Jeśli funkcja main zwróci 0 to znaczy, że program wykonał się poprawnie. Jeśli wartość większą od zera to znaczy, że nastąpił błąd o tym właśnie numerze. Pierwszy parametr funkcji `main`, mówi ile parametrów podał użytkownik po nazwie funkcji. Parametr `argc` jest równy 1 jeśli użytkownik nie podał żadnego parametru, 2, jeśli podał jeden parametr itp. Parametr argc jest większy o 1 od liczby parametrów programu gdyż przy starcie automatycznie jest dodawana pełna nazwa uruchamianego programu (`argv[0]`). Drugi parametr (`argv`) jest tablicą wskaźników wskazujących na kolejne parametry programu. Zerowy element tej tablicy (`argv[0]`) wskazuje na pełną nazwę programu, pierwszy element (`argv[1]`) pokazuje na pierwszy parametr programu itd.

Pierwszą instrukcją funkcji main jest wywołanie funkcji `printf`. Ta funkcja służy do wysyłania sformatowanych wydruków do standardowego urządzenia wyjściowego. Format wydruku określa pierwszy parametr funkcji printf. Jest to łańcuch. Jego znaki są wysyłane do standardowego urządzenia wyjściowego aż do napotkania znaku `%`. Znak `%` oznacza,

że w tym miejscu należy podstawić parametr występujący za formatem (drugi parametr funkcji `printf`). Napotkanie drugie znaku `%` oznacza, że w tym miejscu należy podstawić drugi parametr po formacie (trzeci parametr funkcji `printf`). Po znaku `%` musi nastąpić dodatkowe określenie typu parametru (np. `s` – łańcuch znaków, `d` – liczba całkowita, `u` – liczba całkowita bez znaku, `x` – liczba całkowita zapisana szesnastkowo, `f` – liczba zmiennoprzecinkowa, `c` – znak itp.). W naszym programie, w formacie nie występują znaki `%` dlatego po formacie nie ma żadnych parametrów.

Drugą instrukcją funkcji main jest wykonanie powrotu i zadeklarowania kodu błędu zwracanego przez tą funkcję (tu 0, czyli program wykonał się poprawnie).

W języku C poszczególne instrukcje kończymy znakiem `;` (średnik).

Zadanie 1. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Witaj /*świecie*/!\n");
  return 0;
}
```

2 Kompilacja i konsolidacja

Aby można było skompilować program, należy najpierw go napisać przy użyciu dowolnego edytora tekstowego i zapisać w pliku np. pod nazwą `hello.cpp`. Do kompilacji i konsolidacji programu zostanie użyty darmowy kompilator firmy Borland w wersji 5.5, który można pobrać z serwera firmy Inprise (dawniej Borland) pod adresem:

`ftp://ftpd.inprise.com/download/bcppbuilder/freecommandLinetools.exe.`

Na tym samym serwerze znajduje się również darmowy debugger pracujący w trybie tekstowym:

`ftp://ftpd.inprise.com/download/bcppbuilder/TurboDebugger.exe.`

Kompilację i konsolidację można wykonać w jednym wywołaniu programu `bcc32`. Ponieważ ścieżka do `bcc32` (domyślnie `c:\borland\bcc55\bin`) musi być zadeklarowana w `PATH`, dlatego wystarczy napisać:

```
bcc32 -tWC -Lc:\borland\bcc55\lib -Ic:\borland\bcc55\include hello.cpp
```

Opcja `-tWC` informuje kompilator, że jest to aplikacja konsolowa, opcja `-L` informuje kompilator gdzie znajdują się biblioteki potrzebne do konsolidacji, opcja `-I` informuje kompilator gdzie znajdują się pliki nagłówkowe.

Ten proces można wykonać w dwóch krokach. Kompilacje wykonujemy poleceniem:

```
bcc32 -c -tWC -Ic:\borland\bcc55\include hello.cpp
```

Parametr `-c` oznacza, żeby `bcc32` tylko skompilował podany program. Natomiast konsolidację wykonujemy poleceniem:

```
ilink32 -c -Lc:\borland\bcc55\lib c0x32.obj hello.obj, hello.exe,
                                          hello.map, cw32.lib import32.lib
```

Parametr `-c` oznacza *case sensive* (wrażliwość na wielkość liter), `c0x32.obj` jest kodem startowym dla aplikacji konsolowych, `hello.obj` jest plikiem pośrednim otrzymanym po kompilacji, `hello.exe` jest nazwą pliku wynikowego, `hello.map`, nazwą pliku z symbolami, a `cw32.lib` i `import32.lib` dwiema obowiązkowymi bibliotekami dla aplikacji jednowątkowych.

Można stworzyć dużo krótszą aplikację. Jeśli zamiast biblioteki `cw32.lib` użyjemy `cw32i.lib` to program będzie wymagał obecności biblioteki dynamicznej `cc3250.dll`. Ta biblioteka musi być dostępna w `PATH` (najlepiej ją umieścić w katalogu `c:\windows\system`)

```
ilink32 -c -Lc:\borland\bcc55\lib c0x32.obj hello.obj, hello.exe,
                                         hello.map, cw32i.lib import32.lib
```

Gotową aplikację uruchamiany przez napisanie nazwy (tu: `hello.exe` albo tylko `hello`) pliku wykonywalnego.

3 Automatyzacja kompilacji i konsolidacji

Proces kompilacji i konsolidacji można zautomatyzować. Jest to szczególnie ważne przy większych projektach. Do zautomatyzowania procesu kompilacji i konsolidacji można wykorzystać standardowe narzędzie wchodzące w skład pakietu *FreeCommanLineTools*, czyli `make`. Najprostszy plik dla programu make może mieć postać:

```
hello.exe: hello.cpp
  bcc32 -tWC -Lc:\borland\bcc55\lib -Ic:\borland\bcc55\include hello.cpp
```

W pierwszej linia programista określił jaki jest cel kompilacji (tu: `hello.exe`) a po dwukropku określił jakie pliki wchodzą w skład projektu. W drugiej linii programista określił co należy zrobić aby ten cel osiągnąć. Reguła (pierwsza linia) musi być napisana bez odstępu z lewej strony, a komenda tej reguły mysi być napisana z co najmniej jedną spacją odstępu.

Jeśli piszemy dużo jednomodułowych programów to można pisać jeden uniwersalny `makefile.mak` (to jest domyślna nazwa projektu gdy uruchomimy program `make` bez parametrów). Aby rozróżnić projekty można przy wywołaniu programu `make` definiować stałą (np. o nazwie `SRC`) o wartości równej nazwie kompilowanego programu. W pliku `makefile.mak` można odwoływać się do stałych przez napisanie `$(`*`xxx`*`)` gdzie *`xxx`* jest identyfikatorem stałej.

```
CC=c:\borland\bcc55\bin\bcc32
LIB=c:\borland\bcc55\lib
INC=c:\borland\bcc55\include
OPT=-5 -tWC

$(SRC).exe: $(SRC).cpp
  $(CC) $(OPT) -L$(LIB) -I$(INC) $(SRC).cpp
```

Opcja kompilatora `-5` oznacza, że program wynikowy będzie optymalizowany dla procesora Pentium (i nie będzie działał na wcześniejszych procesorach). Podobny plik dla programu make można napisać gdy zależy nam na rozdzieleniu procesu kompilacji i procesu konsolidacji. Przykładowy plik `hello.mak` może wyglądać następująco.

```
CC=c:\borland\bcc55\bin\bcc32
LINK=c:\borland\bcc55\bin\ilink32
LIB=c:\borland\bcc55\lib
INC=c:\borland\bcc55\include
STDLIB=cw32.lib import32.lib
CCOPT=-5 -v -tWC
LINKOPT=-v

.cpp.obj:
  $(CC) -c $(CCOPT) -I$(INC) $<

hello.exe: hello.obj
  $(LINK) -c $(LINKOPT) -L$(LIB) c0x32.obj hello.obj, hello.exe, hello.map, $(STDLIB)

hello.obj: hello.cpp
```

Opcje `-v` dla kompilatora i dla konsolidatora oznaczają, że powstanie aplikacja z pełną informacją dla debugera. W tym przykładzie zastosowano definicję reguły ogólnej, mówiącej co należy zrobić gdy w programie zaistnieje potrzeba przekształceniu pliku z rozszerzeniem `cpp` na plik z rozszerzeniem `obj`. W komendzie realizującej to przekształcenie zostało użyte makro `$<`, które jest zastępowane przez pełną nazwę przekształcanego pliku. W ostatniej linii mówiącej o konieczności zamiany `hello.cpp` na `hello.obj` nie ma potrzeby pisania komendy przekształcającej gdyż jest ona napisana w regule `.cpp.obj`.

4 Preprocesor

Preprocesor działa na zasadzie wstępnej obróbki tekstu bez wnikania w zasady języka C. Dwie podstawowe dyrektywy preprocesora to:

Dyrektywa	Opis
`#include <`*nazwa pliku*`>`	inkluzja z katalogu kompilatora
`#include "`*nazwa pliku*`"`	inkluzja z katalogu projektu
`#define` *symbol ciąg*	definicja bez parametrów
`#define` *symbol*`(`*parametry*`)` *ciąg*	definicja z parametrami

Rzadziej stosowane dyrektywy to:

Dyrektywa	Opis
`#undef` *`symbol`*	odwołanie definicji

`#if` *`wyrażenie`*
`instrukcje`
`#else`
`instrukcje`
`#endif`
	kompilacja warunkowa

`#ifdef` *`symbol`*
`instrukcje`
`#else`
`instrukcje`
`#endif`
	kompilacja warunkowa

`#ifndef` *`symbol`*
`instrukcje`
`#else`
`instrukcje`
`#endif`
	kompilacja warunkowa

W identyczny sposób definiuje się deklaracje kompilatora. Te dyrektywy nie mają wpływu na preprocesor, są przekazywane do kompilatora i dopiero on na nie reaguje. Dwie ważniejsze dyrektywy kompilatora to:

Dyrektywa	Opis
`#error` *`komunikat`*	przerwanie kompilacji z komunikatem błędu
`#pragma` *`parametry`*	zmiana opcji kompilacji

4.1 Inkluzje

Pierwsza postać inkluzji (`#include <...>`) poszukuje wskazanego pliku w ścieżce standardowej (podanej w parametrach kompilatora) a druga (`#include "..."`) poszukuje w pliku z projektem. Preprocesor wstawi treść wskazanego pliku w miejscu wystąpienia inkluzji.

4.2 Definicje

Definicja polega na tym, że w trakcie czytania pliku z kodem źródłowym programu wszystkie wystąpienia ***`symbolu`*** są zastępowane przez ***`ciąg`***. Symbol zostanie rozpoznany i zastąpiony jeśli będzie odrębnym słowem (a nie częścią innego słowa) oraz nie może być częścią łańcucha. W trakcie zastępowania symbolu ciągiem preprocesor doda spacje (przed i za ciągiem). Nie wszystkie symbole można zdefiniować. Między innymi zastrzeżone symbole to: `__DATE__`, `__TIME__`, `__FILE__`, `__LINE__`. Dwa pierwsze symbole przechowują datę i czas kompilacji, trzeci przechowuje nazwę kompilowanego programu, czwarty numer właśnie kompilowanej linii. Dzięki dwóm ostatnim symbolom można precyzyjnie poinformować użytkownika w jakim module i w której jego linii nastąpił błąd.

Zadanie 2. Co wydrukuje poniższy program?

```
#include <stdio.h>
#define a 10
```

```
int main(int /*argc*/, char **/*argv*/)
{
  printf("dziesieć = a = %d\n",a);
  return 0;
}
```

Zadanie 3. Co wydrukuje poniższy program?

```
#include <stdio.h>
#define suma(a,b) a+b

int main(int /*argc*/, char **/*argv*/)
{
  printf("5*suma(3,8)=%d\n", 5*suma(3,8));
  printf("5*11       =%d\n", 5*11);
  return 0;
}
```

Efekty pracy preprocesora można obejrzeć, gdyż w pakiecie znajduje się program `cpp32.exe` przekształcający tekst programu po eliminacji inkluzji i definicji. Efekt pracy `cpp32.exe` jest zapisywany w pliku z rozszerzeniem `.i`. Analiza pliku pozwoli znaleźć niezamierzone efekty pracy preprocesora.

Zadanie 4. Pokazać błędy kompilacji w poniższym programie.

```
#include <stdio.h>
#define a 10

int main(int /*argc*/, char **/*argv*/)
{
  int a=20;
  printf("a=%d\n",a);
  return 0;
}
```

Zadanie 5. Pokazać błędy kompilacji w poniższym programie.

```
#include <stdio.h>
#define a 10

int main(int /*argc*/, char **/*argv*/)
{
  printf("a+1=%d\n",++a);
  return 0;
}
```

Zadanie 6. Poniższy tekst jest zapisany w pliku `mod1.h`.

```
#include "mod1.h"
```

Skompilować program `mod1.h`.

Zadanie 7. Poniższy tekst jest zapisany w pliku `mod2.h`.

```
#include "mod3.h"
```

Poniższy tekst jest zapisany w pliku `mod3.h`.

```
#include "mod2.h"
```

Skompilować program `mod2.h`.

4.3 Opcje kompilacji

Dyrektywy kompilatora nie generuj kodu ale wpływają na sposób kompilacji programu. Dyrektyw kompilatora jest bardzo dużo. Oto ważniejsze z nich:

Dyrektywa	Opis
`#pragma comment (exestr, "`*`tekst`*`")`	umieszczanie komentarza w pliku wykonywalnym
`#pragma argsused`	ignorowanie ostrzeżenia o nieużywanych parametrach w najbliższej funkcji
`#pragma warn -`*`numer-ostrzeżenia`*	zablokowanie ostrzeżenia
`#pragma warn +`*`numer-ostrzeżenia`*	odblokowanie ostrzeżenia

Na przykład użycie `#pragma warn -8057` ma taki sam skutek jak `#pragma argsused` z tym, że na trwałe.

5 Jednostki leksykalne

W treści programu występują słowa (ciągi znaków bez odstępów i bez ograniczników w środku zwane też literałami), które reprezentują elementy programu. Kompilator rozpoznaje te słowa na podstawie ich budowy.

Jeśli słowo zaczyna się cyfrą różna od 0 (np. 123, 2, 8878, 1.2, 1e3) to jest traktowany jako liczba. Liczba jest całkowita jeśli nie ma części ułamkowej i nie ma części wykładniczej. W przeciwnym wypadku liczba jest traktowana jako wartość rzeczywista.

Jeśli słowo zaczyna się cyfrą zero a drugi znak jest różny od litery `x`, to liczba jest traktowana jako liczba w zapisie ósemkowym (np. 0377 jest liczba całkowitą o wartości 255 w zapisie dziesiętnym).

Jeśli pierwszym znakiem jest zero a drugim mała litera x to jest to liczba całkowita w zapisie szesnastkowym (np. 0x1f jest liczbą całkowitą o wartości 31).

Jeśli słowo zaczyna się znakiem apostrof to jest to stała znakowa (`char`). Stałe znakowe traktowane są jak liczby całkowite ze znakiem (chyba, że zmieni to opcja kompilatora). Znaki są kodowane według tabeli ASCII. Niektóre znaki można uzyskać stosując prefiks \ (*backslash*). Na przykład `\t` to tabulator, `\r` to powrót karetki, `\n` to znak nowej linii, `\` to odwrotny ukośnik, `\'` to apostrof. Po ukośniku można napisać liczbę. Będzie ona traktowana jak kod znaku zapisany ósemkowo (np. `'\101'` to znak `'A'`). Najczęściej koduje się w ten sposób znak *null* (`'\0'`). Można również zdefiniować znak podając kod szesnastkowo. W tym wypadku po ukośniku dajemy znak `x`. (np. `'\x41'` to znak `'A'`).

Jeśli słowo zaczyna się cudzysłowem to program traktuje taki napis jako stałą łańcuchową (*string*). Stała kończy się na drugim cudzysłowie. Stała łańcuchowa to tablica znaków. Kompilator za ostatnim znakiem łańcucha wstawi znak *null* (wartość 0). Aby wewnątrz łańcucha uzyskać znak cudzysłów należy użyć kombinację dwóch znaków: `\"`. Jeśli bezpośrednio po sobie (może być odstęp) nastąpią dwa łańcuchy to kompilator zrobi ich konkatenację.

Wszystkie pozostałe słowo będą traktowane jako symbole (zwane też jednostkami leksykalnymi). Będą to słowa kluczowe, zmienne, nazwy funkcji, operatory itp.

Zadanie 8. Co wydrukuje poniższy program?

```
#include <stdio.h>
```

```
int main(int /*argc*/, char **/*argv*/)
{
  printf("%s\n", "\201 \x61 "
         "a");
  return 0;
}
```

6 Zmienne

Deklarowanie zmiennych polega na rezerwacji miejsca w pamięci komputera na przechowywanie wartości określonego typu. W kompilatorze C są wbudowane podstawowe typy proste: `int` – typ całkowity, `char` – typ znakowy, `float` – typ rzeczywisty pojedynczej precyzji, `double` – typ rzeczywisty podwójnej precyzji.

Oprócz tego można zmieniać podstawowe typy przez dodanie dwóch dodatkowych określeń: `signed` albo `unsigned` oraz `short` albo `long`. Nie zawsze można użyć tych dodatkowych określeń typu. Poniższy program pokazuje ile bajtów jest rezerwowanych na poszczególne typy:

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  printf("char        %d\n", sizeof(char));
  printf("short int   %d\n", sizeof(short int));
  printf("int         %d\n", sizeof(int));
  printf("long int    %d\n", sizeof(long int));
  printf("float       %d\n", sizeof(float));
  printf("double      %d\n", sizeof(double));
  printf("long double %d\n", sizeof(long double));
  return 0;
}
```

W programie użyto specjalne makro (`sizeof`), które zwraca wielkość pamięci zajmowanej przez podany w nawiasie typ. W języku C jest jeszcze operator `sizeof`, który zwraca liczbę bajtów potrzebną na zapamiętanie podanej zmiennej.

6.1 Zmienne globalne i lokalne

Zmienne można deklarować globalnie (poza funkcją `main`) albo lokalnie (wewnątrz funkcji `main` lub innej własnej funkcji). Zmienne globalne są przechowywane w pamięci programu przez cały czas wykonywania programu a zmienne lokalne są tworzone w pamięci komputery przy skoku do funkcji i usuwane z pamięci przy powrocie z funkcji. Wewnątrz funkcji można dowolnie zagnieżdżać bloki (`{...}`). Zmienne zadeklarowane w tych blokach są usuwane na końcu bloku.

```
#include <stdio.h>

int a=1;                  //statyczna zmienna globalna

#pragma argsused
```

```
int main(int argc, char **argv)
{
  printf("a=%d\n", a);

  int a=2;               //automatyczna zmienna lokalna
  printf("a=%d\n", a);
  {
    int a=3;             //automatyczna zmienna lokalna
    printf("a=%d\n", a);
  }
  printf("a=%d\n", a);
  return 0;
}
```

Ten przykład ilustruje jeszcze jedną cechę charakterystyczną dla języka C. Deklaracje inicjowane są traktowane jak instrukcje.

6.2 Zmienne zainicjowane i niemodyfikowalne (ustalone)

Zmienne można inicjować. Po nazwie zmiennej można napisać znak `=` a po nim wartość. Taka deklaracja zmiennej jest równoważna instrukcji podstawienia. Inicjowanie zmiennych globalnych odbywa się przed wywołaniem funkcji `main`. Deklarując zmienne o nadanej wartości początkowej można zabronić ich późniejszego modyfikowania. Przy deklaracji takiej zmiennej (stałej) dodaje się słowo kluczowe `const`.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  const int a=10;
  printf("a=%d\n", a);
  a=a+1;                 //błąd kompilacji, zakaz modyfikowania wartości.
  printf("a=%d\n", a);
  return 0;
}
```

7 Zmienne wskaźnikowe

Zmienne wskazujące otrzymuje się przez dodanie gwiazdki między typem a zmienną (np. deklaracja: `int *wsk` deklaruje zmienną `wsk` przechowującą adres w pamięci operacyjnej, pod którym zapamiętana jest liczba całkowita). Wszystkie zmienne wskazujące zajmują w pamięci komputera 4 bajty.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  printf("char *        %d\n", sizeof(char *));
  printf("short int *   %d\n", sizeof(short int *));
  printf("int *         %d\n", sizeof(int *));
  printf("long int *    %d\n", sizeof(long int *));
  printf("float *       %d\n", sizeof(float *));
  printf("double *      %d\n", sizeof(double *));
```

```
  printf("long double * %d\n", sizeof(long double *));
  return 0;
}
```

Przy operowaniu na zmiennych wskaźnikowych bardzo przydatne są dwa operatory: wyłuskania (dereferencji) oznaczony symbolem `*` i referencji (adresacji) oznaczony symbolem `&`. Jeśli zadeklaruję zmienną wskazującą na liczbę całkowitą (`int *wsk`) to zmienna `wsk` jest adresem a `*wsk` jest wskazywaną wartością całkowitą. Operator referencji to nic innego jak odczytania adresu podanej zmiennej. Na przykład jeśli zadeklaruję zmienną całkowitą `int a` to mogę oczytać adres tej zmiennej w pamięci komputera pisząc `&a`.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a=10;
  int *wska=&a;

  printf("a       %d\n", a);
  printf("adres a %d\n", &a);
  printf("wska    %d\n", wska);
  printf("*wska   %d\n", *wska);
  return 0;
}
```

7.1 Zmienne referencyjne

Zmienne referencyjne są bardzo podobne do zmiennych wskaźnikowych. Deklarując je zamiast `*` dajemy `&`. Zmienne referencyjne tak na prawdę przechowują adres ale gdy chcemy odwołać się do wskazywanego obiektu to nie używamy operatora wyłuskania.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a=10;
  int &refa=a;      //zmienna refa jest tożsama ze zmienną a
  int *wska=&a;     //wska wskazuje na zmienna a

  printf("a     %d\n", a);
  printf("refa  %d\n", refa);
  printf("*wska %d\n", *wska);

  a=a+1;

  printf("a     %d\n", a);
  printf("refa  %d\n", refa);
  printf("*wska %d\n", *wska);

  refa=refa+1;

  printf("a     %d\n", a);
  printf("refa  %d\n", refa);
  printf("*wska %d\n", *wska);
```

```
  *wska=*wska+1;

  printf("a    %d\n", a);
  printf("refa  %d\n", refa);
  printf("*wska %d\n", *wska);
  return 0;
}
```

7.2 Tablice

Tablice deklaruje się podobnie jak pojedyncze zmienne ale za nazwą zmiennej podajemy rozmiar tablicy w nawiasach kwadratowych. Elementy tablicy są zawsze numerowane od zera. Zmienna tablicowa jest tożsama z wskaźnikiem na zerowy element i odwrotnie każdy wskaźnik jest tożsamy z tablicą wskazywanych elementów.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a[5],*c;

  a[0]=10; a[1]=20;
  printf("a[0]=  %d\n", a[0]);
  printf("a[1]=  %d\n", a[1]);
  printf("*a=    %d\n", *a);
  printf("*(a+1)=%d\n", *(a+1));
  c=a;
  printf("c[0]=  %d\n", c[0]);
  printf("c[1]=  %d\n", c[1]);
  printf("*c=    %d\n", *c);
  printf("*(c+1)=%d\n", *(c+1));
  return 0;
}
```

W języku C nie można kopiować całych tablic w instrukcji podstawienia.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a[5],b[5];

  a[0]=10; a[1]=20;
  printf("a[0]=%d, a[1]=%d\n", a[0], a[1]);
  memcpy(b,a,sizeof(a));                    //ale b=a; jest błędne
  printf("b[0]=%d, b[1]=%d\n", b[0], b[1]);
  return 0;
}
```

Zmienne tablicowe można inicjować przez podanie wartości kolejnych elementów podanych wewnątrz nawiasów klamrowych (`{ ... }`) i rozdzielone przecinkami. Jeśli inicjatorów jest mniej niż zadeklarowano to pozostałe elementy są zerowane.

```
#include <stdio.h>
```

```
int main(int /*argc*/, char **/*argv*/)
{
  int a[5]={10,20};

  printf("a[0]=  %d\n", a[0]);
  printf("a[2]=  %d\n", a[2]);
  return 0;
}
```

W przypadku inicjowanych zmiennych tablicowych można pominąć rozmiar tablicy. Zostanie on ustalony na podstawie ilości inicjatorów.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a[]={10,20};      //równoważne: int a[2]=...

  printf("a[0]=%d\n", a[0]);
  return 0;
}
```

W sposób szczególny traktowane są tablice znaków. Jako inicjator tablicy znaków można podać stałą łańcuchową:

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  char a[]="abc";              //równoważne: char a[4]=...
  char b[]={'a', 'b', 'c'};    //równoważne: char b[3]=...

  printf("a=%s\n", a);
  printf("a=%c%c%c\n", a[0],a[1],a[2]);
  printf("b=%s\n", b);                        //tak nie wolno
  printf("b=%c%c%c\n", b[0],b[1],b[2]);    //tak dobrze
  return 0;
}
```

Rozmiar tablicy a jest większy o 1 niż liczba znaków w łańcuchy gdyż kompilator dodaje znak końca łańcucha ('\0').

Język C umożliwia również deklarowanie zmiennych tablicowych wielowymiarowych. Każdy wymiar deklaruje się w odrębnej parze nawiasów kwadratowych. Podobnie przy wołaniu elementów, każdy wymiar podaje w odrębnej parze nawiasów kwadratowych:

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
  int *b=a[0];
  int (*c)[3]=a;

  printf("a[1][1]    =%d\n", a[1][1]);
  printf("*(b+4)     =%d\n", *(b+4));
```

```
  printf("(*(c+1))[1]=%d\n", (*(c+1))[1]);
  printf("c[1][1]    =%d\n", c[1][1]);
  return 0;
}
```

Tablica a jest utożsamiana z adresem zerowego wiersza czyli nie jest tożsame z `int *` tylko `wiersz *` gdzie `wiersz=int[3]` (można to zapisać w jednej linii `int(*)[3]`[1]). Przy używaniu takiego wskaźnika (w powyższym przykładzie jest to zmienna c) istotna jest kolejność wykonywania działań. Określa to tablica priorytetów, która będzie omówiona w rozdziale poświęconym wyrażeniom. Zapis `(*(c+1))[1]` oznacza, że w pierwszej kolejności trzeba dostać się do pierwszego wiersza `*(c+1)` a potem w tym wierszu znaleźć pierwszy element `[1]`. Jeśli zabrakłoby nawiasów okrągłych to napis `*(c+1)[1]` byłby rozumiany jako `*((c+1)[1])` czyli `*(e[1])` gdzie `e=c+1`, a to z kolei byłoby równoważne `*(e+1)` czyli `*(c+2)` czyli wskazanie na drugi wiersz tablicy. Wszystko wynika z tego, że operator dostępu do elementu tablicy `[]` ma wyższy priorytet (zerowy) niż operator dereferencji `*` (pierwszy).

7.3 Struktury

Typy złożone można budować samodzielnie za pomocy słowa kluczowego `struct` i `union`. Struktury to konkatenacja podanych zmiennych. Struktury w języku C deklarujemy pisząc słowo kluczowe `struct`, potem nazwę typu strukturowego, a między nawiasami sześciennymi piszemy listę pól struktury. Lista jest separowana średnikami. Każdy element składa się z nazwy typu i listy identyfikatorów rozdzielonych przecinkami. Po zamykającym nawiasie klamrowym można od razu deklarować zmienne chociaż spotyka się to bardzo rzadko. Separatorem listy zmiennych jest przecinek a kończy ją średnik. Liczbę zespoloną można zadeklarować jako:

```
#include <stdio.h>

struct Complex
{
  double re;
  double im;
};

int main(int /*argc*/, char **/*argv*/)
{
  Complex a;

  a.re=1.0; a.im=2.0;
  printf("%f+i*%f\n", a.re,a.im);
  return 0;
}
```

Równoważna deklaracja struktury `Complex` to:

```
struct Complex
{
  double re, im;
};
```

[1]Nawiasy okrągłe są konieczne. Gdyby ten typ zapisać int *[3] to byłaby to trzyelementowa tablica, która przechowuje wskaźniki liczb całkowitych.

Rozmiar struktury jest równy sumie składowych albo trochę więcej. Czasami kompilator zostawia wolne miejsca między polami tak aby każde pole było wyrównane do 4 bajtów (to zależy od opcji kompilatora). Struktury inicjuje się podobnie jak tablice:

```
#include <stdio.h>

struct Complex
{
  double re;
  double im;
};

int main(int /*argc*/, char **/*argv*/)
{
  Complex a={1.0, 2.0};

  printf("%f+i*%f\n", a.re,a.im);
  return 0;
}
```

Struktury można kopiować w całości w jednej instrukcji przypisania:

```
#include <stdio.h>

struct Complex
{
  double re;
  double im;
};

int main(int /*argc*/, char **/*argv*/)
{
  Complex a={1.0, 2.0};
  Complex b;

  printf("%f+i*%f\n", a.re,a.im);
  b=a;
  printf("%f+i*%f\n", b.re,b.im);
  return 0;
}
```

7.4 Unie

Obok struktur mamy do dyspozycji unie. Unie deklaruje się podobnie jak struktury ale używa się słowa kluczowego `union`. Unia nie jest konkatenacją pól. W unii wszystkie pola zaczynają się od początku unii. Różnicę między unią a strukturą ilustruje poniższy przykład:

```
#include <stdio.h>

struct ComplexS
{
  double re;
  double im;
};
```

```
union ComplexU
{
  double re;
  double im;
};

int main(int /*argc*/, char **/*argv*/)
{
  ComplexS a;
  ComplexU b;

  printf("rozmiar ComplexS %d\n", sizeof(ComplexS));
  printf("rozmiar ComplexU %d\n", sizeof(ComplexU));
  printf("adres zmiennej a typu ComplexS %d\n", &a);
  printf("adres pola re w ComplexS       %d\n", &a.re);
  printf("adres pola im w ComplexS       %d\n", &a.im);
  printf("adres zmiennej b typu ComplexU %d\n", &b);
  printf("adres pola re w ComplexU       %d\n", &b.re);
  printf("adres pola im w ComplexU       %d\n", &b.im);
  return 0;
}
```

7.5 Zmienne wyliczeniowe

Zmienne wyliczeniowe to zmienne przyjmujące wartości określone przez identyfikatory. Identyfikatory podaje programista w specjalnej deklaracji `enum`. Deklaracja wyliczeniowa składa się ze słowa kluczowego `enum` oraz listy identyfikatorów wartości umieszczonej w nawiasach klamrowych. Separatorem listy jest przecinek. Identyfikatorom są przydzielane kolejne wartości całkowite poczynając od zera. Po identyfikatorze wartości można umieścić znak `=` a ponim jawnie podać wartość identyfikatora. Następny identyfikator dostanie wartość o 1 większą.

```
#include <stdio.h>

enum {poniedzialek, wtorek, sroda=3, czwartek, piatek, sobota, niedziela};

int main()
{
  printf("poniedzialek=%d",poniedzialek);
  printf("czwartek    =%d",czwartek);
  return 0;
}
```

Wartości poszczególnych indentyfikatorów nie muszą być unikalne.

Zadanie 9. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  const int a=10;
  int *wsk = (int *)&a;
```

```
  printf("a=   %d\n", a);
  printf("*wsk=%d\n", *wsk);
  *wsk = *wsk+1;
  printf("a=   %d\n", a);
  printf("*wsk=%d\n", *wsk);
  return 0;
}
```

Zadanie 10. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  const int a[5]={10,20,30,40,50};
  int *wsk = (int *)&a;

  printf("a[0]=%d\n", a[0]);
  printf("*wsk=%d\n", *wsk);
  *wsk = *wsk+1;
  printf("a[0]=%d\n", a[0]);
  printf("*wsk=%d\n", *wsk);
  return 0;
}
```

Zadanie 11. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  char a[3]="abc";

  printf("%s\n", a);
  return 0;
}
```

Zadanie 12. Co wydrukuje poniższy program?

```
#include <stdio.h>

enum {poniedzialek, wtorek, sroda=3, czwartek, piatek=2, sobota, niedziela};

int main()
{
  printf("środa =%d",sroda);
  printf("sobota=%d",sobota);
  return 0;
}
```

7.6 Klasy

Klasy są związane z programowanie obiektowym. Są podobne do struktur ale deklaruje się je słowem kluczowym `class`. Ponieważ programowanie obiektowe zrobiło w ostatnich czasach ogromną karierę dlatego klasom będzie poświęcony odrębny rozdział.

8 Typy

Deklarowanie typów jest podobne do deklarowania zmiennych. Jeśli deklaracje zmiennej poprzedzimy słowem kluczowym `typedef` to kompilator potraktuje tą deklarację jako deklarację typu a nie deklarację zmiennej, a nazwa zmiennej stanie się nazwą typu. Oczywiście inicjatory w deklaracjach typu nie są dopuszczalne. Liczby zespolone można zadeklarować w sposób następujący:

```
typedef struct
{
  double re;
  double im;
} Complex;
```

Dwuwymiarową tablicę można zadeklarować w następujący sposób:

```
#define N 10
typedef double Macierz[N][N];
Macierz Jakobian;
```

W drugiej linii zadeklarowano typ tablicowy o nazwie `Macierz`, a w trzeciej linii zadeklarowano zmienną o nazwie `Jakobian`, która jest dwuwymiarową tablicą liczb podwójnej precyzji o rozmiarze 10×10.

9 Wyrażenia

Specyfiką języka C jest traktowanie wyrażeń jako instrukcji. Wyrażenie występujące jako instrukcja jest opracowywane (wyliczane) a wynik jest ignorowany. Najdziwniejszym rozwiązaniem w języku C jest potraktowanie instrukcji przypisania jako wyrażenia przypisania:

```
#include <stdio.h>

#pragma argsused
int main(int argc, char **argv)
{
  int a,b;

  a=1;                          //ignoruję wynik wyrażenia, który jest równy 1
  printf("a=%d\n", a);
  b=a=2;                        //ignoruję wynik wyrażenia, który jest równy 2
  printf("a=%d, b=%d\n", a,b);
  return 0;
}
```

Najtrudniejszym zagadnieniem przy budowaniu wyrażeń w języku C są priorytety operatorów. Jest ich dużo bo dużo jest operatorów. Tabela przedstawia operatory w kolejności malejących priorytetów:

<table>
<tr><th>Priorytet</th><th>Grupa operatorów</th><th>Operator</th><th>Opis</th></tr>
<tr><td>0</td><td>dostępu</td><td>()
[]
.
->
::
Type()</td><td>operator wołania funkcji
odwołanie do elementu tablicy
odwołanie do pola struktury
odwołanie do pola struktury
kwalifikator zakresu
rzutowanie na typ (Type)</td></tr>
<tr><td>1</td><td>jednoargumentowe</td><td>++
--
!
~
+
&
*
sizeof
new
delete</td><td>inkrementacja
dekrementacja
negacja
negacja bitowa
liczba dodatnia
referencja
dereferencja (wyłuskanie)
rozmiar zmiennej
alokacja pamięci
dealokacja pamięci</td></tr>
<tr><td>2</td><td>rzutowanie typów</td><td>(Type)</td><td>rzutowanie na typ (Type)</td></tr>
<tr><td>3</td><td>dostępu</td><td>.*
->*</td><td></td></tr>
<tr><td>4</td><td>multiplikatywne</td><td>*
/
%</td><td>mnożenie całkowite i zmiennoprzecinkowe
dzielenie całkowite i zmiennoprzecinkowe
reszta z dzielenia</td></tr>
<tr><td>5</td><td>addytywne</td><td>+
-</td><td>dodawanie
odejmowanie</td></tr>
<tr><td>6</td><td>bitowe</td><td><<
>></td><td>przesuwanie bitów w lewo
przesuwanie bitów w prawo</td></tr>
<tr><td>7</td><td>relacyjne</td><td><
<=
>
>=</td><td>mniejsze
mniejsze lub równe
większe
większe lub równe</td></tr>
<tr><td>8</td><td>relacyjne</td><td>==
!=</td><td>równe
różne</td></tr>
<tr><td>9</td><td>bitowe</td><td>&</td><td>koniunkcja bitów</td></tr>
<tr><td>10</td><td>bitowe</td><td>^</td><td>alternatywa wykluczająca bitów</td></tr>
<tr><td>11</td><td>bitowe</td><td>|</td><td>alternatywa bitów</td></tr>
<tr><td>12</td><td>boolowskie</td><td>&&</td><td>koniunkcja</td></tr>
<tr><td>13</td><td>boolowskie</td><td>||</td><td>alternatywa</td></tr>
<tr><td>14</td><td>warunkowe</td><td>?:</td><td>operator warunkowy</td></tr>
<tr><td>15</td><td>przypisania</td><td>=
*=
/=
%=
+=
-=
<<=
>>=
&=
^=
|=</td><td>przypisanie
domnożenie i przypisanie
podzielenie i przypisanie
obliczenie reszty i przypisanie
dosumowanie i przypisanie
odjęcie i przypisanie
przesunięcie bitów w lewo i przypisanie
przesunięcie bitów w prawo i przypisanie
koniunkcja bitowa i przypisanie
alternatywa wykluczająca i przypisanie
alternatywa i przypisanie</td></tr>
<tr><td>16</td><td>połączenia</td><td>,</td><td>łączenie wyrażeń w jedno wyrażenie</td></tr>
</table>

Typy można konwertować na inne za pomocą operatorów rzutowania. Oba operatory: *Type*`()`, `(`*Type*`)` działają identycznie tylko w pierwszym przypadku typ musi być jednym słowem a w drugim może być wieloczłonowy.

```
#include <stdio.h>

#pragma argsused
int main(int argc, char **argv)
{
  int a=-1;
  unsigned int b;

  b=(unsigned int)a;
  printf("a=%d\n", a);
  printf("b=%u\n", b);
  b=unsigned(a);           //ale b=unsigned int(a); spowoduje błąd kompilacji
  printf("a=%d\n", a);
  printf("b=%u\n", b);
  return 0;
}
```

Operatory inkrementacji następnikowej i poprzednikowej ilustruje poniższy przykład:

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a;
  a=10;
  printf("a=%d\n", a);
  printf("++a=%d\n", ++a);
  printf("a=  %d\n", a);
  a=10;
  printf("a=%d\n", a);
  printf("a++=%d\n", a++);
  printf("a=%d\n", a);
  return 0;
}
```

Operacja negacji bitowe (`~`) polega zanegowaniu każdego bita z osobna. Operacje bitowe dwuargumentowe (`&`, `^`, `|`) są wykonywane parami na poszczególnych bitach operandów.

Negacja (`!`) zmienia liczbę różna od zera na zero i zero na liczbę różną od zera. Koniunkcja i alternatywa (`&&`, `||`) są wykonywane tak jak rachunku zdań przy czym fałszem jest wartość zerowa a prawdą dowolna wartość różna od zera.

Ciekawostką języka C jest to, że operatory przypisania (`=`, `*=`, `/=` itp.) są operatorami a nie instrukcjami jak w innych językach programowania.

Operator warunku jest jedynym operatorem trójargumentowym: `a ? b : c`. Jeśli `a` jest różne od zera to wynikiem jest `b`. W przeciwnym wypadku wynikiem jest `c`.

Operator połączenia pozwala wpisać kilka wyrażeń w miejscu gdzie dozwolone jest tylko jedno wyrażenie.

Zadanie 13. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  int a=10;

  printf("%d\n", +/*x*/+a);
  printf("%d\n", ++a);
  return 0;
}
```

10 Instrukcje

W języku C jest bardzo wiele instrukcji. Kompilator rozpoznaje instrukcję na podstawie słowa kluczowego, które ją zaczyna ale są także instrukcje, które nie mają swojego słowa kluczowego: instrukcja opracowania wyrażenia, instrukcja pusta, instrukcja grupująca, instrukcja deklaracyjna.

10.1 Instrukcja pusta

Instrukcja pusta składa się z pustego napisu i średnika:

```
  ;
```

Instrukcję pustą używamy w przypadkach gdy nic nie chcemy zrobić np.

```
#include <stdio.h>

void copy(char *dest, char *src)
{
#pragma warn -8060
  while (*(nazwa++)=*(param++))
    ;
#pragma warn +8060
}

int main(int /*argc*/, char **argv)
{
  char nazwa[100];

  copy(nazwa,argv[0]);
  printf("%s\n", nazwa);
  return 0;
}
```

Powyższy program przekopiuje nazwę programu do zmiennej nazwa. Wewnątrz pętli `while` jest instrukcja pusta, gdyż właściwe kopiowanie odbywa się przy okazji sprawdzania warunku kontynuacji pętli. Ponieważ kompilator C podejrzewa nas o chęć porównaniu dwóch liczba a nie podstawienia dlatego generowane jest ostrzeżenie o numerze 8060. Ponieważ operator przypisania jest tu użyty świadomie dlatego dyrektywa kompilatora `#pragma warn` wyłącza to ostrzeżenie na czas kompilacji pętli `while`.

10.2 Instrukcja opracowania wyrażenia

W skrócie jest nazywana instrukcją wyrażeniową. Jest to najczęściej spotykana funkcja. Działanie tej instrukcji jest bardzo nietypowe, gdyż wynik jej działania jest ignorowany. Trwałe są skutki uboczne. Na przykład skutkiem ubocznym opracowania jest zapamiętanie wartości w zmiennej:

```
(a=10)+20;
```

Wynikiem tego wyrażenia jest liczba 30 ale ten wynik jest ignorowany. Skutkiem ubocznym jest to, że po opracowaniu tego wyrażenia zmienne `a` ma wartość 10. Również wołanie funkcji lub procedury jest traktowane jako wyrażenie, a zatem jako instrukcja. Tak często spotykane w tych materiałach wołanie funkcji `printf` jest w istocie opracowywaniem wyrażenia. Wynik działania funkcji `printf` (liczba wydrukowanych znaków) jest w tych przykładach ignorowany.

10.3 Instrukcja grupująca

Jest to bardzo ważna instrukcja gdyż pozwala wykonać wiele instrukcji w sytuacjach gdy dopuszczalne jest wykonanie tylko jednej instrukcji. Na przykład pętla `while` dopuszcza wykonanie tylko jednej instrukcji. Instrukcja grupująca składa się z nawiasu sześciennego otwartego (`{`), listy instrukcji i nawiasu sześciennego zamkniętego (`}`). Instrukcja grupująca występuje również przy definiowaniu funkcji.

Instrukcja grupująca pod jednym względem jest inna od pozostałych: nie kończy się średnikiem.

10.4 Instrukcja deklaracyjna

Instrukcja deklaracyjna to deklarowanie zmiennych. W przeciwieństwie do innych języków programowania, deklarowanie zmiennych potraktowano jako czynność do wykonania a nie tylko jako informację dla kompilatora aby zarezerwował miejsce w pamięci komputera do przechowywania wartości określonego typu. W rzeczywistości kompilator rezerwuje miejsce na zmienne jeśli tylko jest to możliwe na etapie kompilacji. Na przykład w ten sposób rezerwowane są wszystkie niezainicjowane zmienne globalne. Aby oszczędzić czas wszystkie niezainicjowane zmienne lokalne w funkcjach są rezerwowane jednym poleceniem a nie wieloma poleceniami, po jednym na każdą zmienną. Podobnie globalne zmienne ustalone (zadeklarowane ze słowem kluczowym `const`) są umieszczone w specjalnym segmencie danych zainicjowanych, który jest częścią pliku wykonywalnego (EXE).

Ogólnie deklarowanie zmiennych ma postać: *`zestaw-specyfikatorów lista-deklaracyjna`*;. Zestaw specyfikatorów to określenie typu i rodzaju zmiennej. Można użyć typu predefiniowanego np. `int`, `double`, `float`, `char`, itp, Można zmienić go modyfikatorami `long`, `short`, `unsigned`, `signed` itp. Do specyfikatorów należą również słowa kluczowe: `static` – deklarowanie zmiennej statycznej, `auto` – deklarowanie zmiennej automatycznej, `register` – deklarowanie zmiennej rejestrowej, `volatile` – deklarowanie zmiennej ulotnej, `extern` – deklarowanie zmiennej zewnętrznej.

10.5 Instrukcja skoku bezwarunkowego

Instrukcja skoku bezwarunkowego jest bardzo rzadko używana. Deklaruje się ją słowem kluczowym `goto`.

```
goto etykieta;
```

Powoduje skok do instrukcji wkazanej przez etykietę. Etykiety można umieścić przed dowolną instrukcją. Etykietę od instrukcji separujemy znakiem : (dwukropek). Skoki bezwakunkowe są cecha charakterystyczną wszelkich asemblerów. W językach wysokiego poziomu unikamy używania skoków bezwarunkowch. Zostały one wyperte przez istrukcje strukturalne takie jak: instrukcja warunkowa (`if`), instrukcje pętli (`for`, `while`, `do`), instrukcje selekcji (`select`). Wiele skoków bezwarunkowych można również zastąpić stosując instrukcje zaniecania (`break`), kontynuacji (`continue`), powrotu (`return`);

10.6 Instrukcja warunkowa

Zadaniem instrukcji warunkowej jest wykonanie jednej z dwóch instrukcji w zależności od wartości warunku. Warunkiem jest wyrażenie, którego wartość 0 jest traktowana jak fałsz a wartość różna od 0 jest traktowana jak prawda. Ogólna budowa instrukcji warunkowej ma postać:

```
if (exp) inst1; else inst2;
```

Jeśli nic nie trzeba robić gdy warunek jest fałszywy to można pominąć słowo kluczowe `else` i instrukcję zanim występującą:

```
if (exp) inst1;
```

Instrukcja *`inst1`* zostanie wykonana gdy *`exp`* jest różne od 0. Gdy *`exp`* jest równe 0 to zostanie wykonana instrukcja *`inst2`*.

Jeśli do wykonania jest kilka czynności to musimy je umieścić we wnętrzu instrukcji grupującej (`{...}`).

10.7 Instrukcja selekcji

Zadaniem instrukcji selekcji jest sprawdzenie wielu warunków i wykonanie wszystkich instrukcji po pierwszym spełnieniu warunku. Ogólna budowa instrukcji warunkowej ma postać:

```
switch (exp)
{
  case const1: inst1a; inst1b; ...
  case const2: inst2a; inst2b; ...
    ⋮
  default: insta; instb; ...
}
```

Jeśli nic nie trzeba robić gdy żaden z warunków nie jest spełniony to można pominąć słowo kluczowe `default` i instrukcję zanim występującą.

Gdy wyrażenie *exp* jest równe stałej *const1* to zostaną wykonane instrukcje: *inst1a*, *inst1b*, ..., *inst2a*, *inst2b*, ..., *insta*, *instb*, ... Jeśli wyrażenie *exp* jest równe stałej *const2* to zostaną wykonane instrukcje: *inst2a*, *inst2b*, ..., *insta*, *instb*, ... Jeśli żaden warunek nie zostanie spełniony to zostaną wykonane tylko instrukcje: *insta*, *instb*, ...

Działanie instrukcji selekcji ilustroje poniższy przykład:

```
#include <stdio.h>

void wybieraj(int n)
{
  switch(n)
  {
  case 1: printf("Wykonuje dla 1\n");
  case 2: printf("Wykonuje dla 2\n");
  case 3: printf("Wykonuje dla 3\n");
  default: printf("Wykonuje dla pozostalych\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  printf("n=1\n"); wybieraj(1);
  printf("n=2\n"); wybieraj(2);
  printf("n=3\n"); wybieraj(3);
  printf("n=4\n"); wybieraj(4);
  return 0;
}
```

Instrukcje występujące w poszczególnych przypadkach nie muszą być ujmowane w nawiasy klamrowe.

10.8 Instrukcja pętli while

Jest to jedna z instrukcji pętli. Ogólna budowa pętli `while` jest następująca:

`while (`*exp*`)` *inst*;

Najpierw opracowywane jest wyrażenie *exp*. Jeśli jest równe 0 (traktowane jako fałsz) to instrukcja się kończy. Jeśli jest różne od 0 (czyli jest prawdziwe) to wykonywana jest instrukcja *inst*. Po wykonaniu instrukcji obliczany jest warunek *exp* i w zależności od jego wartości instrukcja `while` się kończy albo ponownie jest wykonywana instrukcja *inst*.

Cechą charakterystyczną pętli `while` jest to, że ciało pętli (*inst*) może nie zostać ani razu wykonane.

Zadanie 14. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  while(0)
    printf("Wykonanie ciała pętli\n");
  return 0;
}
```

10.9 Instrukcja pętli do

Drugą z instrukcji pętli jest pętla `do`. Jej budowa jest następująca:

`do` ***inst***`;` `while (`*exp*`);`

W przeciwieństwie do pętli `while`, najpierw wykonywane jest ciało pętli (***inst***) a dopiero potem jest sprawdzany warunek kontynuacji pętli. Ciało pętli będzię wykonane ponownie gdy warunek jest prawdziwy (czyli wyrażenie ***exp*** jest różny od 0).

Instrukcję pętli `do` można zapisać za pomocą pętli `while` w sposób następujący:

```
inst;
while(exp)
  inst;
```

Zadanie 15. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  do
    printf("Wykonanie ciała pętli\n");
  while(0);
  return 0;
}
```

10.10 Instrukcja pętli for

Trzecią z instrukcji pętli jest pętla `for`. Ma ona następującą budowę:

`for (`*inst1*`;` *exp*`;` *inst2*`)` ***inst3***`;`

Najpierw program wykona instrukcję inicjującą pętlę ***inst1***. Następnie opracuje wyrażenie ***exp***. Jeśli wartość wyrażenia będzie 0 to cała instrukcja `for` się zakończy. Jeśli wartość będzie różna od 0 (czyli prawda) to nastąpi wykonanie ciała pętli ***inst3***. Po wykonaniu ciała pętli nastąpi wykonanie instrukcji ***inst2***, która najczęściej służy do zmiany wartości licznika pętli. Kolejną czynnością będzie opracowanie wyrażenia ***exp***, które zadecyduje czy należy już zakończyć instrukcję `for` (gdy jest 0), czy ponownie wykonać ciało pętli ***inst3*** (gdy jest różne od 0). Instrukcja pętli `for` dopuszcza aby warunek kontynuacji pętli (***exp*** został pominięty. W takim przypadku domniemywa się, że (***exp*** jest różne od zera (czyli jest prawdą).

Instrukcję pętli `for` można zapisać za pomocą pętli `while` w sposób następujący:

```
inst1;
while(exp)
{
  inst3;
  inst2;
}
```

Poniższy przykład pozwala prześledzić kolejność wykonywanych działań w instrukcji pętli `for`

```
#include <stdio.h>

#pragma argsused
int main (int argc, char **argv)
{
  int i;

  for (printf("init\n"),i=0; printf("exp i=%d\n",i),i<3; printf("inc i=%d\n",i),i++)
  {
    printf("body i=%d\n",i);
  }
  return 0;
}
```

Zadanie 16. Co wydrukuje poniższy program?

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  for (;;)
    printf("Wykonanie ciała pętli\n");
  return 0;
}
```

Zadanie 17. Wskazać błędy kompilacji w poniższych programach.

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  while()
    printf("Wykonanie ciała pętli\n");
  return 0;
}
```

```
#include <stdio.h>

int main(int /*argc*/, char **/*argv*/)
{
  do
    printf("Wykonanie ciała pętli\n");
  while();
  return 0;
}
```

10.11 Instrukcja kontynuacji

Instrukcja kontynuacji wykonuje skok bezwarunkowy za ostatnią instrukcję ciała bieżącej instrukcji pętli (`while`, `do`, `for`) ale przed końcem pętli. Ta instrukcja słada się tylko ze słowa kluczowego `continue`:

```
continue;
```

Instrukcję kontynuacji stosuje się wszędzie tam, gdy w określonych warunkach trzeba pominąć końcową część ciała pętli lub ciała innej instrukcji.

Zadanie 18. Pokazać błędy kompilacji.

```
#include <stdio.h>

void proc(int n)
{
  if (n)
  {
    printf("przed continue\n");
    continue;
    printf("po continue\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  proc(0);
  proc(1);
  return 0;
}
```

Zadanie 19. Pokazać błędy kompilacji.

```
#include <stdio.h>

void proc(int n)
{
  switch (n)
  {
  case 1:
    printf("przed continue\n");
    continue;
    printf("po continue\n");
  default:
    printf("default\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  proc(0);
  proc(1);
  return 0;
}
```

Zadanie 20. Co wydrukuje poniższy program?

```
#include <stdio.h>

#pragma argsused
int main (int argc, char **argv)
{
  int n=3;

  while (n--)
  {
    printf("przed continue n=%d\n",n);
    if (n>1)
      continue;
    printf("po continue n=%d\n",n);
  }
  return 0;
}
```

10.12 Instrukcja zaniechania

Instrukcja zaniechania wykonuje skok bezwarunkowy do pierwszej instrukcji za bieżącym blokiem instrukcji selekcji (`switch`), albo instrukcji pętli (`while`, `do`, `for`). Ta instrukcja składa się tylko ze słowa kluczowego `break`:

```
break;
```

Najczęściej wykorzystuje się tą instrukcję wewnątrz instrukcji selekcji (`switch`).

```
#include <stdio.h>

void wybieraj(int n)
{
  switch(n)
  {
  case 1: printf("Wykonuje dla 1\n"); break;
  case 2: printf("Wykonuje dla 2\n"); break;
  case 3: printf("Wykonuje dla 3\n"); break;
  default: printf("Wykonuje dla pozostalych\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  printf("n=1\n"); wybieraj(1);
  printf("n=2\n"); wybieraj(2);
  printf("n=3\n"); wybieraj(3);
  printf("n=4\n"); wybieraj(4);
  return 0;
}
```

Zadanie 21. Co wydrukuje poniższy program?

```
#include <stdio.h>

void wybieraj(int n)
{
  switch(n)
  {
  case 1:
    {
      printf("Wykonuje dla 1a\n");
      break;
      printf("Wykonuje dla 1b\n");
    }
    printf("Wykonuje dla 1c\n");
  case 2:
    printf("Wykonuje dla 2\n"); break;
  default:
    printf("Wykonuje dla pozostalych\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  printf("n=1\n"); wybieraj(1);
  printf("n=2\n"); wybieraj(2);
  printf("n=3\n"); wybieraj(4);
  return 0;
}
```

Zadanie 22. Co wydrukuje poniższy program?

```
#include <stdio.h>

void wybieraj(int n)
{
  switch(n)
  {
  case 1:
  case 2:
    if (n==1) { printf("Wykonuje dla %d\n",n); break; }
    printf("Wykonuje dla %d\n",n);
    break;
  default:
    printf("Wykonuje dla pozostalych\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  printf("n=1\n"); wybieraj(1);
  printf("n=2\n"); wybieraj(2);
  printf("n=3\n"); wybieraj(4);
  return 0;
}
```

Zadanie 23. Pokazać błędy kompilacji.

```
#include <stdio.h>

void proc()
{
   printf("przed break\n");
   break;
   printf("po break\n");
}

#pragma argsused
int main (int argc, char **argv)
{
  proc();
  return 0;
}
```

Zadanie 24. Pokazać błędy kompilacji.

```
#include <stdio.h>

void proc(int n)
{
  if (n)
  {
    printf("przed break\n");
    break;
    printf("po break\n");
  }
}

#pragma argsused
int main (int argc, char **argv)
{
  proc(0);
  proc(1);
  return 0;
}
```

10.13 Instrukcja powrotu

Instrukcja powrotu wykonuje skok bezwarunkowy za ostatnią instrukcję funkcji i procedury, co oznacza zakończenie funkcji i procedry. Ta instrukcja składa się słowa kluczowego `return` i ewentualnie wyrażenia:

`return;`

`return` *exp*`;`

Pierwsza forma służy do wyjścia z procedury a druga forma do wyjścia z funkcji. Funkcje muszą mieć intrukcję powrotu gdyż jest to jedyny sposób na podanie wyniku zwracanego przez funkcję.

11 Definiowane funkcji

W języku C można definiować własne funkcje. W pierwszej wersji języka były wyłącznie funkcje, potem wprowadzono słowo kluczowe `void`, które pozwala definiować procedury. Definicja funkcji rozpoczyna się typem zwracanego wyniku (albo `void`), potem deklaruje się nazwę funkcji i listę parametrów formalnych w nawiasach okrągłych. Lista parametrów formalnych to lista par *typ nazwa*, które separujemy znakiem przecinek. Po nawiasie zamykającym listę parametrów umieszczamy instrukcję grupującą. Funkcję obliczającą największy wspólny dzielnik dwóch liczba całkowitych można zdefiniować w sposób następujący:

```
#include <stdio.h>

int nwd(int n, int m)
{
  int r;

  r=n % m;
  while(r)
  {
    n=m;
    m=r;
    r=n % m;
  }
  return m;
}

int main(int /*argc*/, char **/*argv*/)
{
  printf("%d\n",nwd(3*6,3*9));
  return 0;
}
```

Do powrotu z funkcji służy instrukcja powrotu, która składa się ze słowa kluczowego `return` i wyrażenia będącego wynikiem działania funkcji. W procedurach używamy słowa `return` bez wyrażenia. W procedurach można pominąć `return` jeśli jest ostatnią instrukcją procedury. Poniżej zdefiniowana jest procedura drukująca liczbę zespoloną:

```
#include <stdio.h>

void print_complex(double re, double im)
{
  printf("%f+i*%f",re,im);
}

int main(int /*argc*/, char **/*argv*/)
{
  print_complex(1.0,2.0);
  return 0;
}
```

Parametry do procedur można przekazywać na dwa sposoby: przez wartość albo przez referencję. Dwa powyższe przykłady pokazują jak przekazać parametry przez wartość.

Kompilator wyliczy wartość parametru aktualnego (przy wywołaniu) i obliczoną wartość przekaże do procedury. Przekazywanie parametrów przez referencję oznacza, że tak naprawdę przekazuje się adres zmiennej. Jeśli funkcja zmodyfikuje taki parametr to po powrocie z funkcji zmienna będzie miała zmienioną wartość. Kompilator traktuje parametr przekazany przez referencję jeśli między typem a identyfikatorem parametru wystąpi znak `&`. Konsekwencją przekazania przez referencję, jest to, że przy wywoływaniu funkcji parametrem aktualnym musi być nazwa zmiennej (nie może to być wyrażenie) albo wyrażenie wskazujące na zmienną takiego typu. Przy przekazywaniu przez referencję, typy parametrów formalny i aktualnych muszą być identyczne.

```
#include <stdio.h>

void print_complex(double re, double im)
{
  printf("%f+i*%f",re,im);
}

void reset_complex(double &re, double &im)
{
  re=0.0;
  im=0.0;
}

int main(int /*argc*/, char **/*argv*/)
{
  double a=1.0, b=2.0;

  print_complex(a,b); printf("\n");
  reset_complex(a,b);
  print_complex(a,b); printf("\n");
  return 0;
}
```

Trzeba uważać na przekazywanie tablic. Poniższy przykład wygląda jak przekazanie przez wartość. W istocie jest to przekazanie przez wartość ale nie tablicy tylko wskaźnika typu `double *`. Program przekazuje adres zerowego elementu tablicy, dlatego podstawienie wewnątrz funkcji ma skutek po powrocie z funkcji.

```
#include <stdio.h>

void print_matrix(double a[3])
{
  printf("%f\n",a[0]);
  printf("%f\n",a[1]);
  a[0]=3.0;
}

int main(int /*argc*/, char **/*argv*/)
{
  double a[4]={1.0, 2.0};

  print_matrix(a);
  print_matrix(a);
  return 0;
}
```

Ten program jest równoważny programowi zamieszczonemu niżej:

```
#include <stdio.h>

void print_matrix(double *a)
{
  printf("%f\n",*a);        //albo  printf("%f\n",a[0]);
  printf("%f\n",*(a+1));    //albo  printf("%f\n",a[1]);
  *a=3.0;                   //albo  a[0]=3.0;
}

int main(int /*argc*/, char **/*argv*/)
{
  double a[4]={1.0, 2.0};

  print_matrix(a);
  print_matrix(a);
  return 0;
}
```

Natomiast struktury, unie i klasy są przekazywane przez wartość jak typy proste.

```
#include <stdio.h>

struct Complex
{
  double re;
  double im;
};

void print_complex(Complex a)
{
  printf("%f+i*%f\n",a.re,a.im);
  a.re=0.0; a.im=0.0;
}

int main(int /*argc*/, char **/*argv*/)
{
  Complex a={1.0, 2.0};
  print_complex(a);
  print_complex(a);
  return 0;
}
```

12 Przysłanianie

W języku C nie ma przysłaniania. Nie można zadeklarować dwóch identyfikatorów należących do tej samej grupy. Natomiast można zadeklarować dwa takie same identyfikatory należące dwóch różnych grup. Na przykład można zadeklarować zmienną i etykietę o tej samej nazwie ale nie można zadeklarować zmiennej i funkcji o tej samej nazwie. Przysłanianie wynika z kolejności poszukiwania identyfikatora w poszczególnych grupach. Najpierw przeszukiwana jest grupa etykiet, potem grupa nazw pól struktur i unii, na koniec grupa nazw zmiennych i funkcji. Każdy blok dostaje swoje grupy identyfikatorów, które są przeszukiwane w pierwszej kolejności.

```
#include <stdio.h>

int a=1;

int main(int /*argc*/, char **/*argv*/)
{
  int a=2;
  {
    int a=3;
    printf("a=%d\n",a);
    printf("a=%d\n",::a);
  }
  printf("a=%d\n",a);
  printf("a=%d\n",::a);
  return 0;
}
```

Zadanie 25. Pokazać błędy kompilacji.

```
#include <stdio.h>

int a()
{
  return 1;
}

int a=2;

#pragma argsused
int main(int argc, char **argv)
{
  printf("a()=%d\n", a());
  printf("a=%d\n", a);
  return 0;
}
```

Zadanie 26. Pokazać błędy kompilacji.

```
#include <stdio.h>

int a()
{
  return 1;
}

#pragma argsused
int main(int argc, char **argv)
{
  int a=2;

  printf("a()=%d\n", a());
  printf("a=%d\n", a);
  return 0;
}
```

13 Odczytywanie parametrów programu

Poniższy program ilustruje sposób odczytania pełnej nazwy uruchomionego programu:

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Nazywam sie: %s\n",argv[0]);
  return 0;
}
```

Parametr `argv` jest tablicą wskaźników wskazujących kolejne parametry programu. W zerowym elemencie jest przechowywana nazwa programu. W formacie (`printf`) umieszczamy pole `%s`, które zostanie zastąpione przez drugi parametr funkcji `printf` (tu: `argv[0]`). Jeśli program zapiszemy w pliku `toja.cpp` to można go skompilować poleceniem: `make -DSRC=toja`.

Drugi przykład ilustruje jak można po kolei odczytać wszystkie parametry (tym razem z pominięciem nazwy programu).

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;

  printf("Liczba parametrów: %d\n",argc-1);
  for (i=1; i<argc; i++)
    printf("parametr nr %d jest równy: %s\n",i,argv[i]);
  return 0;
}
```

Zadanie 27. Napisać powyższy program i zachować go w pliku `dane.cpp`. Skompilować program poleceniem `make -DSRC=dane` a następnie przetestować z następującymi parametrami:

```
dane
dane a
dane aa bb
dane aa,bb
dane aa, bb
dane "aa bb"
dane 'aa bb'
dane c:\"Program Files"\"Common Files"\Proof c:\"My Documents"
dane "c:\Program Files\Common Files\Proof" "c:\My Documents"
dane -c pr1.cpp
dane /c pr1.cpp
```

Zadaniem następnego programu będzie obliczenie logarytmu dziesiętnego z podanej liczby rzeczywistej.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
  int i;
  double a;
  char *blad;

  if (argc!=2)
  {
    printf("Program wymaga podania jednej liczb rzeczywistej\n");
    return 1;
  }
  a=strtod(argv[1],&blad);
  if (*blad!='\0')
  {
    printf("Parametr nie jest porawną liczbą rzeczywistą\n");
    return 2;
  }
  if (a<=0.0) then
  {
    printf("Parametr musi być liczbą rzeczywistą większą od zera\n");
    return 2;
  }

  printf("Log10(%f)=%f\n",a,log10(a));
  return 0;
}
```

Funkcje matematyczne są zgromadzone w pliku nagłówkowym `math.h`. Funkcja obliczająca logarytm dziesiętny nazywa się `log10`. Parametry podane przez użytkownika po nazwie programu są tekstowe. Jeśli chcemy przeprowadzać operacje numeryczne na tych parametrach to musimy dokonać konwersji łańcucha znaków na liczbę zmiennoprzecinkową (typ `double`). Konwersję można dokonać standardową funkcją `strtod`, która jest zadeklarowana w pliku nagłówkowym `stdlib.h`. Ta funkcja zwraca wartość po konwersji. Wymaga trzech parametrów: pierwszy parametr to konwertowany łańcuch, drugi to wskaźnik miejsca błędu. Jeśli nie było błędu to wskaźnik błędu wskazuje na znak o kodzie 0 (`'\0'`). Jeśli był błąd to wskaźnik błędu pokazuje na literę, która spowodowała błąd konwersji.

Jeśli program zapiszemy w pliku `log.cpp` a następnie skompilujemy poleceniem `make -DSRC=log` to logarytm dziesiętny z 1.5 będzie można obliczyć poleceniem:

```
log 1.5
```

Kolejny przykład pokazuje jak odczytać parametry, które są liczbami całkowitymi. Zadaniem tego programu będzie znalezienie największego wspólnego podzielnika dwóch liczb całkowitych.

```
#include <stdio.h>
#include <stdlib.h>

/////////////////////////////////////////////////////////////
```

```
// funkcja obliczająca nawiększy wspólny dzielnik dwóch liczb całkowitych
int nwd(int n, int m)
{
  int r;

  r=n % m;
  while(r)
  {
    n=m;
    m=r;
    r=n % m;
  }
  return m;
}

/////////////////////////////////////////////////////////////
// główny funkcja programu, która odczytuje parametry, zamienia je na liczy całkowite,
// sprawdza czy mają poprawne wartości, wołająca funkcję obliczającą NWD i
// drukująca wynik obliczeń.
int main(int argc, char **argv)
{
  int i;
  int a,b;
  char *blad;

  if (argc!=3)
  {
    printf("Program wymaga podania dwóch liczb całkowitych\n");
    return 1;
  }
  a=strtol(argv[1],&blad,10);
  if (*blad!='\0')
  {
    printf("Pierwszy parametr nie jest poprawną liczbą całkowitą\n");
    return 2;
  }
  if (a<=0)
  {
    printf("Pierwszy parametr musi być liczbą całkowitą większą od zera\n");
    return 2;
  }

  b=strtol(argv[2],&blad,10);
  if (*blad!='\0')
  {
    printf("Drugi parametr nie jest poprawną liczbą całkowitą\n");
    return 2;
  }
  if (b<=0)
  {
    printf("Drugi parametr musi być liczbą całkowitą większą od zera\n");
    return 2;
  }

  printf("Największy wspólny dzielnik liczb %d i %d to: %d\n",a,b,nwd(a,b));
  return 0;
}
```

Obliczenia największego wspólnego dzielnika odbywają się w funkcji `nwd()`. Ta funkcja dostaje dwa parametry całkowite i zwraca liczbę całkowitą równą największemu wspólnemu dzielnikowi podanych argumentów. Funkcja jest niezależna od systemu operacyjnego gdyż nie posiada żadnych operacji wejścia/wyjścia. Funkcja `nwd()` zakłada, że argumenty są poprawne. Kontrola poprawności danych wejściowych odbywa się w głównej funkcji programu (`main`).

14 Optymalizowanie kodu

Język C daje ogromne możliwości optymalizowania kodu źródłowego. Oprócz tego kompilator potrafi optymalizować kod wynikowy na poziomie rozkazów mikroprocesora. Tym nie mniej programista może również wydatnie wpłynąć na optymalność kodu wynikowego. Poniżej pokazane są dwie funkcje obliczające wartość średnią. Algorytm obliczania jest identyczny, różnica jest tylko w kodowaniu.

```
//////////////////////////////////////////////////////////////
// funkcja obliczająca wartość średnią w sposób Pascalowy
double srednia(int n, double a[])
{
  int i;
  double s;

  s=0.0;
  for (i=0; i<n; i++)
    s=s+a[i];
  return s/n;
}

//////////////////////////////////////////////////////////////
// funkcja obliczająca wartość średnią zgodnie z filozofią języka C
double srednia(int n, double *a)
{
  double s=0.0;
  while (n--)
    s += *(a++);
  return s/n;
}
```

Drugi przykład dotyczy operacji macierzowych. Można się umówić, że macierz kwadratowa to wektor składający z następujących po sobie wierszy macierzy. Poniżej przedstawione są dwie równoważne funkcje realizujące dokładnie to samo mnożenie macierzy kwadratowej przez wektor kolumnowy.

```
//////////////////////////////////////////////////////////////
// Funkcja mnożącą macierz przez wektor w sposób Pascalowy.
// Dane wejściowe to wymiar macierzy i wektora (N) oraz
// macierz (a) i wektor (b). Wynik zostanie umieszczony
// w wektorze (c) o wymiarze N.
void matrix_mul_vector(int N, double *a, double *b, double *c)
{
  int i,j;
  double s;
```

```
  for (i=0; i<N; i++)
  {
    s=0.0;
    for (j=0; j<N; j++)
      s = s + a[i*N+j] * b[j];
    c[i]=s;
  }
}

////////////////////////////////////////////////////////////
// Funkcja mnożąca macierz przez wektor zgodnie z filozofią języka C
// Parametry jak wyżej.
void matrix_mul_vector(int N, double *a, double *b, double *c)
{
  int i,j;
  double *p;

  i=N;
  while (i--)
  {
    *c=0.0;
    p=b;
    j=N;
    while (j--)
      *c += *(a++) * *(p++);
    c++;
  }
}
```

15 Programowanie obiektowe

Programowanie obiektowe nie jest wielką rewolucją w technikach programowania. Jest to raczej ewolucja. Oprogramowanie obiektowe integruje dane i procedury uprawnione do operowania na tych danych. Ta integracja odbyła się przez rozszerzenie składni struktur. Taką rozszeżoną strukturę nazywamy klasą lub typem obiektowym. W języku C++ w strukturze można teraz deklarować dane i funkcje. Dane zadeklarowane w klasie nazywamy atrybutami klasy a funkcje (procedury) nazywamy metodami klasy.

15.1 Deklarowanie klas

W języku C++ wprowadzono dodatkowe słowo kluczowe `class`, które słyży do deklarowania klas. Jest ono prawie tożsame ze słowem `struct`. Wyjątek dotyczy traktowania pól. W klasach przez domniemanie składniki są prywatne a w strukturach są publiczne.

W celu zwiększenia bezpieczeństwa pracy zespołowej nad wieloma klasami wprowadzono dodatkowe zakresy widoczności atrybutów i metod. Słowo kluczowe `public:` wewnątrz klasy oznacza, że wszystko co teraz zostanie zadeklarowane jest dostępne dla wszystkich. Słowo kluczowe `private:` wewnątrz klasy oznacza, że wszystko co teraz zostanie zadeklarowane jest dostępne tylko dla metod tej klasy. Słowo kluczowe `protected:` wewnątrz klasy oznacza, że wszystko co teraz zostanie zadeklarowane jest dostępne tylko dla metod tej klasy i dla jej potomków.

```
class nazwa
```

```
{
private:
  deklaracje atrybutów i metod prywatnych
protected:
  deklaracje atrybutów i metod chronionych
public:
  deklaracje atrybutów i metod publicznych
}
```

Poszczególne zakresy widoczności (`private`, `protected`, `public`) mogą się powtarzać wielokrotnie i mogą występować w dowolnej kolejności. Poniższy przykład pokazuje jak zadeklarować liczby zespolone w wersji obiektowej. Klasa będzie znała wartość części rzeczywistej i urojonej oraz będzie umiła wydrukować wartość liczby zespolonej oraz policzyć moduł liczby zespolonej.

```
#include <stdio.h>
#include <math.h>

class CComplex
{
public:
  double m_re;
  double m_im;

  void  print();
  double mod();
};

void CComplex::print()
{
  printf("%f+i*%f\n", m_re,m_im);
}

double CComplex::mod()
{
  return sqrt(m_re*m_re+m_im*m_im);
}

int main(int /*argc*/, char **/*argv*/)
{
  CComplex a;

  a.m_re=1.0; a.m_im=2.0;
  a.print();
  printf("modul a=%f\n", a.mod());
  return 0;
}
```

Jedna z tradycji programistycznych (Visual C++ firmy Microsoft) nakazuje poprzedzanie nazywy klasy literą `C` (od słowa *class*) oraz poprzedzania atrybutów klasy literami `m_` (litera m i podkreślenie co pochodzi od słowa *member*).

15.2 Konstruktory i dekstruktory

Przedstawiona w poprzednim rozdziale definicja klasy `CComplex` nie była zbyt „porządna”. Aby zainicjować wartości atrybutów klasy trzeba było dokonać jawnych podstawień (np. `a.m_re=1.0`). To zmusiło programistę do upublicznienia atrybutów `m_re` i `m_im`. Programując obiektowo staramy się ukryć wewnętrzną budowę klasy. Pozwalamy odwoływać się do atrybutów tylko za pośrednictwem metod.

Do inicjowania pól obiektu służą wyróżnione metody zwane konstruktorami. Konstruktor ma taką samą nazwę jak klasa. Konstruktorów może być kilka ale muszą się one różnić listą parametrów. Konstruktory są procedurami, których nie wolno deklarować za pomocą słowa kluczowego `void`. Każda klasa bez konstruktora ma wbudowany konstruktor bezparametrowy, który zeruje wszystkie atrybuty. Jak widać w przeciwieństwie do zwykłych zmiennych, klasy są zawsze zainicjowane.

Wersja programu z konstruktorem może wyglądać następująco.

```
#include <stdio.h>
#include <math.h>

class CComplex
{
  double m_re;
  double m_im;
public:
  CComplex(double re, double im);
  void  print();
  double mod();
};

CComplex::CComplex(double re, double im)
{
  m_re=re;
  m_im=im;
}

void CComplex::print()
{
  printf("%f+i*%f\n", m_re,m_im);
}

double CComplex::mod()
{
  return sqrt(m_re*m_re+m_im*m_im);
}

int main(int /*argc*/, char **/*argv*/)
{
  CComplex a=CComplex(2.0,1.0);
  CComplex b(1.0,2.0);

  a.print();
  printf("modul a=%f\n", a.mod());
  b.print();
  printf("modul a=%f\n", b.mod());
  return 0;
}
```

Mając do dyspozycji konstruktor możemy ukryć atrybuty (umieścić je w sekcji `private`). Teraz można inicjować obiekty klasy `CComplex` na dwa sposoby: używając jawnie inicjatora (znak `=` i nazwa konstruktora z parametrami), albo w wersji skróconej przez napisanie tylko listy parametrów. Odpowiedni konstruktor zostanie dobrany na podstawie listy parametrów aktualnych.

Konstrutora nie można użyć ponownie dla już utworzonego obiektu[2]. Dlatego trzeba przewidzieć zwykłą metodę dostępu do atrybutów klasy.

⋮

```
class CComplex
{
  double m_re;
  double m_im;
public:
  CComplex(double re, double im);
  void set(double re, double im);
  void  print();
  double mod();
};

void set(double re, double im)
{
  m_re=re;
  m_im=im;
}
```

⋮

Bardzo krótkie metody (np. takie jake konstruktor i metoda `set` w klasie `CComplex`) można zadeklarować jako metody otwarte (*inline*). Takie metody nie są wywoływane przez skok do funkcji tylko są w całości wkompilowywane w miejscach ich wołania. To sprawia, że wynikowy kod programu jest dłuższy ale za to trochę szybciej wyknywany. Klasa `CComplex` z metoda otwartymi będzie wyglądać następująco.

```
#include <stdio.h>
#include <math.h>

class CComplex
{
  double m_re;
  double m_im;
public:
  CComplex(double re, double im) { m_re=re; m_im=im; }
  void set(double re, double im) { m_re=re; m_im=im; }
  void  print();
  double mod();
};

void CComplex::print()
{
  printf("%f+i*%f\n", m_re,m_im);
}
```

[2]To wynika z faktu, że konstruktor nie tylko inicjuje pola obiektu ale również inicjuje tablicę skoków wirtualnych (patrz polimorfizm) a ona nie może być inicjowania wielokrotnie.

```
double CComplex::mod()
{
  return sqrt(m_re*m_re+m_im*m_im);
}

    ⋮
```

W głównej funkcji programu następuje wołanie dwukrotnie metody `mod()` najpierw dla obiektu `a`, a potem dla obiektu `b`. Rodzi się pytanie skąd metoda `mod()` wie, które `m_re` i `m_im` użyć jeśli w ciele metody nie ma jawnego odwołania do obiektu. Otóż w trakcie wołania metody (np. `a.mod()`) do metody przekazywany jest niejawny parametr o nazwie `this`, który przechowuje adres obieku, na rzecz którego ta metoda została wywołana. Poniżej zapisana jest metoda `mod()` jako zwykła funkcja i fragment głównej funkcji która woła taką metodę zamienioną na funkcję.

```
double mod(CComplex *wsk)}
{
  return sqrt(wsk->m_re*wsk->m_re+wsk->m_im*wsk->m_im);
}
    ⋮
int main()
{
    ⋮
  printf("modul a=%f\n", mod(&a));
    ⋮
```

Zamiast zmiennej `this`[3] w powyższej funkcji występuje zmienna `wsk`.

Destruktory to funkcje wołane, gdy zmienna jest usuwana z pamięci komputera. Destruktory definiuje się podobnie jak konstruktory tylko nazwę klasy trzeba poprzedzić znakiem ~ (tylda). Poniższy przykład ilustroje kiedy są wołane destruktory i konstruktory. Jedna zmienna jest globalna, druga lokalną zmienna głównej funkcji, a trzecia jest tworzona dynamicznie.

```
#include <stdio.h>
#include <math.h>

class CComplex
{
  double m_re;
  double m_im;
public:
  CComplex(double re, double im);
  ~CComplex();
};

CComplex::CComplex(double re, double im)
{
  m_re=re;
  m_im=im;
```

[3]Identyfikator `this` jest słowem kluczowym, i nie można go użyć.

```
  printf("Inicjuję po utworzeniu    %f+i*%f\n", m_re,m_im);
}

CComplex::~CComplex()
{
  printf("Sprzątam przed usunięciem %f+i*%f\n", m_re,m_im);
}

CComplex a(1.0,1.0);

int main(int /*argc*/, char **/*argv*/)
{
  printf("Zaczynam main\n");
  CComplex *c;
  CComplex b(2.0,2.0);

  c=new CComplex(3.0,3.0);

  // tu coś robię na obiektach a,b,c

  delete c;
  return 0;
}
```

15.3 Atrybuty i metody statyczne

W języku C++ można zadeklarować atrybuty, które są wspólne dla wszystkich obiektów. Takie atrybuty nazywa się statycznymi i deklaruje się słowem kluczowym `static`. Pola statyczne nie wliczają się do rozmiaru obiektu gdyż dla wszystkich obiektów, ile by ich nie było, pole statyczne jest tylko jedno. Na przykład gdyby zaszła potrzeba zliczania obiektów to zamiast robić oddzielny liczni i martwić się o jego „ręczną” aktualizację możemy zadeklarować taki licznik jako atrybut statyczny i aktualizować go w konstruktorze i destruktorze.

```
#include <stdio.h>
#include <math.h>

class CComplex
{
  double m_re;
  double m_im;
  static int m_licznik;
public:
  CComplex(double re, double im) { m_re=re; m_im=im; m_licznik++; }
  ~CComplex() { m_licznik--; }
  int LiczbaObiektow() { return m_licznik; }
};

int CComplex::m_licznik=0;  //deklarowanie i inicjowanie pola statycznego

CComplex a(1.0,1.0);

int main(int /*argc*/, char **/*argv*/)
{
  CComplex *c;
```

```
  CComplex b(2.0,2.0);

  printf("Liczba obiektów=%d\n",a.LiczbaObiektow());
  c=new CComplex(3.0,3.0);
  printf("Liczba obiektów=%d\n",a.LiczbaObiektow());
  delete c;
  return 0;
}
```

Jak widać deklaracja klasy nie rezerwuje miejsca w pamięci komputera na pole statyczne. Trzeba to zrobić samodzielnie piszą typ zmiennej, nazwę klasy, nazwę pola i ewentualnie wyrażenie inicjujące po znaku `=`. Jeśli inicjator zostanie pominięty to takie pole będzie miało wartość 0. Bez względu, który obiekt zostanie zapytany o liczbę obiektów, zostanie zwrócona globalna liczba obiektów.

Powyższy program, choć formalnie poprawny, nie jest optymalnie napisany. Nieoptymalność dotyczy metody `LiczbaObiektów()`. Po co przekazywać do takiej metody wskaźnik bieżącego obiektu (ukryty parametr `this`) skoro taka metoda nie korzysta z takiego adresu? Metodę, która korzysta wyłącznie z atrybutów statycznych można również uczynić metodą statyczną. Robi się to przez dodanie słowa kluczowego `static`.

⋮

```
class CComplex
{
  double m_re;
  double m_im;
  static int m_licznik;
public:
  CComplex(double re, double im) { m_re=re; m_im=im; m_licznik++; }
  ~CComplex() { m_licznik--; }
  static int LiczbaObiektow() { return m_licznik; }
};
```

⋮

Po tej zmianie nie muszę już specyfikować obiektu na rzecz, którego ta metoda statyczna działa. Wystarczy napisać nazwę klasy i po `::` (dwa dwukropki) nazwę metody statycznej.

⋮

```
int main(int /*argc*/, char **/*argv*/)
{
  CComplex *c;
  CComplex b(2.0,2.0);

  printf("Liczba obiektów=%d\n",CComplex::LiczbaObiektow());
  c=new CComplex(3.0,3.0);
  printf("Liczba obiektów=%d\n",CComplex::LiczbaObiektow());
  delete c;
  return 0;
}
```

⋮

15.4 Dziedziczenie

Dziedziczenie to jedna z nowości programowania obiektowego. Na bazie jednej klasy mamy możliwość zbudowani nowej klasy definiując tylko te aspekty nowej klasy, które się zmieniły w stosunku do starej klasy. Ta cecha programownia obiektowego jest szczególnie ceniona przez twórców gotowych bibliotek klas gdyż nie muszą oni udostępniać źródeł swoich klas tylko po to, aby przyszli użytkownicy mogli zmieniać ich funkcjonalność. Po prostu na bazie dostarczonej klasy buduje się nową klasę i zmienia tylko to co użytkownikowi nie odpowiada.

Klasę pochodną definiuje się według ogólnego wzoru:

```
class nazwa : widoczność klasa-bazowa
{
private:
  deklaracje atrybutów i metod prywatnych
protected:
  deklaracje atrybutów i metod chronionych
public:
  deklaracje atrybutów i metod publicznych
}
```

W miejsce ***widoczność*** można wpisać słowo kluczowe `public` albo `private`. Pierwsze oznacza, że dziedziczone akrytuby i metody zachowują swoje zakresy widoczności, natomiast drugie oznacza, że wszystkie stają sie na zewnątrz niedostępne bez względu na zakres widoczności w klasie bazowej. Zakresy widoczności można przeanalizować na przykładzie następującego programu:

```
#include <stdio.h>

////////////////////////////
class A
{
private:
  int m_a;
protected:
  int m_b;
public:
  int m_c;
public:
  A(int a, int b, int c);
  void inc();
};

A::A(int a, int b, int c)
{
  m_a=a; m_b=b; m_c=c;
}

void A::inc()
{
  m_a++;
  m_b++;
```

```
  m_c++;
}

////////////////////////////
class B : private A
{
private:
  int m_d;
protected:
  int m_e;
public:
  int m_f;
public:
  B(int a, int b, int c, int d, int e, int f);
  void inc();
};

B::B(int a, int b, int c, int d, int e, int f) : A(a,b,c)
{
  m_d=d; m_e=e; m_f=f;
}

void B::inc()
{
  //brak dostępu do m_a, bo jest to prywatny atrybut klasy bazowej
  m_b++;
  m_c++;
  m_d++;
  m_e++;
  m_f++;
}

////////////////////////////
class C : public A
{
private:
  int m_d;
protected:
  int m_e;
public:
  int m_f;
public:
  C(int a, int b, int c, int d, int e, int f);
  void inc();
};

C::C(int a, int b, int c, int d, int e, int f) : A(a,b,c)
{
  m_d=d; m_e=e; m_f=f;
}

void C::inc()
{
  //brak dostępu do m_a, bo jest to prywatny atrybut klasy bazowej
  m_b++;
  m_c++;
  m_d++;
```

```
  m_e++;
  m_f++;
}

/////////////////////////////////
#pragma argsused
int main(int argc, char **argv)
{
  A a(1,2,3);
  B b(1,2,3,4,5,6);
  C c(1,2,3,4,5,6);

  //brak dostępu do m_a bo jest to prywatny atrybut klasy A.
  //brak dostępu do m_b bo jest to zabezpieczony atrybut klasy A.
  printf("a=(?,?,%d)\n",a.m_c);
  //brak dostępu do m_a, m_b, m_c bo było dziedziczenie z ukrytą klasą bazową.
  //brak dostępu do m_d bo jest to prywatny atrybut klasy B.
  //brak dostępu do m_e bo jest to zabezpieczony atrybut klasy B.
  printf("b=(?,?,?,?,?,%d)\n",b.m_f);
  //brak dostępu do m_a bo jest to prywatny atrybut klasy bazowej A.
  //brak dostępu do m_b bo jest to zabezpieczony atrybut klasy bazowej A.
  //brak dostępu do m_d bo jest to prywatny atrybut klasy B.
  //brak dostępu do m_e bo jest to zabezpieczony atrybut klasy B.
  printf("c=(?,?,%d,?,?,%d)\n",c.m_c,c.m_f);

  return 0;
}
```

W tym przykładzie są zadeklarowane trzy klasy. Klasą bazową jest klasa `A`. Z niej powstaje klasa `B` przez dziedziczenie z ukrytą klasą bazową, oraz klasa `C` przez dziedziczenie z jawną klasą bazową. Obiekty klasy `A` zajmują 12 bajtów w pamięci komputera (trzy liczby całkowite: `m_a`, `m_b`, `m_c`). Obiekty klasy `B` i `C` zajmują 24 bajty w pamięci komputera (sześć liczb całkowitych: `m_a`, `m_b`, `m_c`, `m_d`, `m_e`, `m_f`).

Jak widać sposób dziedziczenia nie ma wpływu na dostęp metod pochodnych do atrybutów klasy bazowej. To co było prywatne jest niedostępne a to co było zabezpieczone i publiczne jest dostępne. Natomiast jest wpływ na dostęp z zewnątrz: to co było publiczne w klasie bazowej (dostępne) przestaje być dostępne przy dziedziczeniu z ukrytą klasą bazową

Aby lepiej zrozumieć potrzebę dziedziczenie proponuję zdefiniować klasy, które bedą potrzebne do napisania (w bliżej nieokreślonej przyszłości) programu graficznego do tworzenia dwuwymiarowych rysunków. Ten program będzie przechowywał obiekty widoczne na rysunku jako figury geometryczne (np. odcinki, linie łamane, prostokąty, wielokąty, koła, elipsy itp.). Każda figura będzie znała swoje położenie w umownych jednostkach (np. w milimetrach względem umownego początku współrzędnych). Każda z tych figur będzie potrafiła się narysować[4]. Proponuję zacząć od trzech najprostszych figur:

Odcinek:	opisany przez wspołrzędne początku i współrzędne końca.
Prostokąt:	opisany przez wspołrzędne środka i dwie długość boków.
Elipsa:	opisana przez wspołrzędne środka i dwa promienie.

W przyszłości można wyposażyć te figury w dodatkowe atrybuty: kolor, grubość linii, sposób wypełnienia itp. Jak widać, we wszystkich figurach powtarzają się współrzędne

[4]W miarę rozwoju programu będzie można dopisywać nowe cechy figur

początku (dwie liczby rzeczywiste). Wspólne atrybuty umieszczamy w klasie bazowej.

```
class CShape
{
protected:
  float m_x;
  float m_y;
public:
  CShape(float x, float y) { m_x=x; m_y=y; }
  void Draw() {}
};
```

Przewiduję, że metody obiektu potomnego będą miały prawo odwołać się bezpośrednio do współrzędnych początku dlatego zostały umieszczone w sekcji `protected`. Metoda `Draw()` nic nie wykonuje bo nie można narysować punktu o zerowym rozmiarze. Na bazie klasy `CShape` budujemy klasy pochodne czyli:

```
class CLine:public CShape
{
  float m_x2;
  float m_y2;
public:
  CLine(float x, float y, float x2, float y2) : CShape(x,y)
  {
    m_x2=x2;
    m_y2=y2;
  }
  void Draw()
  {
    printf("Rysuję odcinek od punktu (%f, %f) do punktu (%f, %f)\n",
           m_x,m_y,m_x2,m_y2);
  }
};

class CRectangle:public CShape
{
  float m_dx;
  float m_dy;
public:
  CRectangle(float x, float y, float dx, float dy) : CShape(x,y)
  {
    m_dx=dx;
    m_dy=dy;
  }
  void Draw()
  {
    printf("Rysuję prostokąt o środku (%f, %f) i wielkości %f, %f\n",
           m_x,m_y,m_dx,m_dy);
  }
};

class CEllipse:public CShape
{
  float m_rx;
  float m_ry;
public:
```

```
  CEllipse(float x, float y, float rx, float ry) : CShape(x,y)
  {
    m_rx=rx;
    m_ry=ry;
  }
  void Draw()
  {
    printf("Rysuję elipsę o środku (%f, %f) i promieniach (%f, %f)\n",
           m_x,m_y,m_rx,m_ry);
  }
};
```

Na uwagę zasługuje sposób wyłania konstruktora klasy bazowej. Po nagłówku a przed ciałem funkcji dajem znak : (dwukropek) a po nim nazwę konstrukora klasy bazowej i listę parametrów aktualnych. W celu przetestowania klas można napisać główną procedurę, która zadeklaruje trzy zmienne o klasach `CLine`, `CRectangle`, `CEllipse` i wywoła dla nich metody `Draw()`.

```
#pragma argsused
int main(int argc, char **argv)
{
  CLine line(10,10,30,40);
  CRectangle rectangle(50,50,25,35);
  CEllipse ellipse(40,40,10,20);

  line.Draw();
  rectangle.Draw();
  ellipse.Draw();

  return 0;
}
```

Każdy z tych obiektów (`CLine`, `CRectangle`, `CEllipse`) będzie zajmował w pamięcie komputera 16 bajtów (cztery liczby typu `float`). Kompilator wie, którą metodę `Draw()` wywołać, gdyż znane są typy zmiennych. Na przykład zmienna `line` jest typu `CLine` czyli dla tej zmiennej zostanie wywołana metoda `CLine::Draw()`.

Wersja z obiektami dynamicznymi niczego tu nie zmieni.

```
#pragma argsused
int main(int argc, char **argv)
{
  CLine *line;
  CRectangle *rectangle;
  CEllipse *ellipse;

  line=new CLine(10,10,30,40);
  rectangle=new CRectangle(50,50,25,35);
  ellipse=new CEllipse(40,40,10,20);

  line->Draw();
  rectangle->Draw();
  ellipse->Draw();

  delete line;
  delete rectangle;
```

```
  delete ellipse;

  return 0;
}
```

15.5 Polimorfizm

Obiekty graficzne (owe figury) będą występować na rysunku wielokrotnie. Można oczywiście zdefiniować wiele tablic, po jednej na każdy typ figury, ale takie rozwiązanie będzie bardzo niewygodne dla programisty. Każdą operację na figurach (np. rysowanie, szukanie prostokąta otaczającego wszystkie figury, skalowanie figur, poszukiwanie figur widocznych na podanym obszrze itp.) trzeba będzie wykonywać na każdej tablicy osobno. Programowanie obiektowe daje możliwość zadeklarowania jednej tablicy, która przechowuje adresy obiektów bazowych.

Gdyby zmienić program główny tak jak pokazano niżej:

```
#pragma argsused
int main(int argc, char **argv)
{
  CShape *figury[10];
  int i;

  figury[0]=new CLine(10,10,30,40);
  figury[1]=new CRectangle(50,50,25,35);
  figury[2]=new CEllipse(40,40,10,20);

  for (i=0; i<3; i++)
    figury[i]->Draw();

  for (i=0; i<3; i++)
    delete figury[i];

  return 0;
}
```

to program byłby błędny gdyż kompilator dla zmiennych `figury[i]` wywoła metodę `CShape::Draw()`. Niezwykłe jest to, że programowanie obiektowe zezwala na podstawienie adresu obiektu potomnego od zmiennej wskazującej na obiek klasy bazowej. Aby program był poprawny należy zamienić metodę `Draw()` na metodę wiertualną. Róźca między zwykłą metodą a metodą wirtualną jest taka, że adres zwykłej metody jest ustalany w trakcie kompilacji. Natomiast w przypadku metod wirtualnych adres procedury jest ustalany w trakcie wykonania programu. Wywołanie metody wirtualnej jest tylko trochę wolniejsze niż wywołanie zwykłej metody. Dochodzi odczytanie adresu skoku z tak zwanej tablicy metod wirtualnych.

Aby zadekralować metodę jako wirtualną, trzeba deklarację metody poprzedzić słowem kluczowym `virtual`. Trzeba to zrobić w klasie bazowej. W klasach potomnych nawet jeśli zapomnimy o słowie `virtual` to i tak wszystkie metody o tej samej nazwie i tej samej liście parametrów będą wirtualne.

```
class CShape
{
protected:
```

```
  float m_x;
  float m_y;
public:
  CShape(float x, float y) { m_x=x; m_y=y; }
  virtual void Draw() {}
};

class CLine:public CShape
{
  float m_x2;
  float m_y2;
public:
  CLine(float x, float y, float x2, float y2) : CShape(x,y)
  {
    m_x2=x2;
    m_y2=y2;
  }
  virtual void Draw()
  {
    printf("Rysuję odcinek od punktu (%f, %f) do punktu (%f, %f)\n",
           m_x,m_y,m_x2,m_y2);
  }
};

class CRectangle:public CShape
{
  float m_dx;
  float m_dy;
public:
  CRectangle(float x, float y, float dx, float dy) : CShape(x,y)
  {
    m_dx=dx;
    m_dy=dy;
  }
  virtual void Draw()
  {
    printf("Rysuję prostokąt o środku (%f, %f) i wielkości %f, %f\n",
           m_x,m_y,m_dx,m_dy);
  }
};

class CEllipse:public CShape
{
  float m_rx;
  float m_ry;
public:
  CEllipse(float x, float y, float rx, float ry) : CShape(x,y)
  {
    m_rx=rx;
    m_ry=ry;
  }
  virtual void Draw()
  {
    printf("Rysuję elipsę o środku (%f, %f) i promieniach (%f, %f)\n",
           m_x,m_y,m_rx,m_ry);
  }
};
```

Po tych zmianach, główna funkcja programu zadziała poprawnie. Nie ma sensu deklarowanie wirtualnych funkcji otwartych (`inline`) gdyż adresy funkcji wirtualnych muszą być umieszczone w tablicy skoków. Dlatego bardziej poprawna deklaracja klas i definicja metod będzię następująca:

```
class CShape
{
protected:
  float m_x;
  float m_y;
public:
  CShape(float x, float y) { m_x=x; m_y=y; }
  virtual void Draw();
};

class CLine:public CShape
{
  float m_x2;
  float m_y2;
public:
  CLine(float x, float y, float x2, float y2) : CShape(x,y)
  {
    m_x2=x2;
    m_y2=y2;
  }
  virtual void Draw();
};

class CRectangle:public CShape
{
  float m_dx;
  float m_dy;
public:
  CRectangle(float x, float y, float dx, float dy) : CShape(x,y)
  {
    m_dx=dx;
    m_dy=dy;
  }
  virtual void Draw();
};

class CEllipse:public CShape
{
  float m_rx;
  float m_ry;
public:
  CEllipse(float x, float y, float rx, float ry) : CShape(x,y)
  {
    m_rx=rx;
    m_ry=ry;
  }
  virtual void Draw();
};

void CShape::Draw()
{
}
```

```
void CLine::Draw()
{
  printf("Rysuję odcinek od punktu (%f, %f) do punktu (%f, %f)\n",
         m_x,m_y,m_x2,m_y2);
}

void CRectangle::Draw()
{
  printf("Rysuję prostokąt o środku (%f, %f) i wielkości %f, %f\n",
         m_x,m_y,m_dx,m_dy);
}

void CEllipse::Draw()
{
  printf("Rysuję elipsę o środku (%f, %f) i promieniach (%f, %f)\n",
         m_x,m_y,m_rx,m_ry);
}
```

16 Metody i klasy abstrakcyjne

Jeśli w przyszłym programie nigdy nie pojawi się obiekt klasy `CShape`, a jest to bardzo prawdopodobne, można uniknąć definiowana pustej metody `CShape::Draw`. Tą metodę można zadeklarować jako metodę abstrakcyjną. W tym celu po nagłówku metody umieszczamy znak `=` (równa się) a po nim 0 (zero).

```
class CShape
{
protected:
  float m_x;
  float m_y;
public:
  CShape(float x, float y) { m_x=x; m_y=y; }
  virtual void Draw()=0;
};
```

Z dalszej części programu usuwamy definicję motody `CShape::Draw()`.

Klasa, która ma chociaż jedną metodę abstrakcyjną jest nazywana klasą abstrakcyjną. Kompilator wykaże błąd jeśli spróbujemy zadeklarować obiekt klasy abstrakcyjnej.

Po tych wszystkich zmianach uszlachetniających bardzo prosto obliczyć najmniejszy prostokąt otaczający wszystkie figury. Dzieki metodom wirtualnym bedzię to można zrobić w jednej pętli operującej na tablicy niejednorodnych obiektów (ale o wspólnej klasie bazowej). W tym celu wzbogacamy klasę bazową o metodę `Box`, która będzie zwracać prostokąt otaczający pojedynczą figurę. Będzie to oczywiście metoda wirtualna.

```
#include <stdio.h>

class CShape
{
protected:
  float m_x;
  float m_y;
public:
```

```
  CShape(float x, float y) { m_x=x; m_y=y; }
  virtual void Draw()=0;
  virtual void Box(float &x1, float &y1, float &x2, float &y2);
};

class CLine:public CShape
{
  float m_x2;
  float m_y2;
public:
  CLine(float x, float y, float x2, float y2) : CShape(x,y)
  {
    m_x2=x2;
    m_y2=y2;
  }
  virtual void Draw();
  virtual void Box(float &x1, float &y1, float &x2, float &y2);
};

class CRectangle:public CShape
{
  float m_dx;
  float m_dy;
public:
  CRectangle(float x, float y, float dx, float dy) : CShape(x,y)
  {
    m_dx=dx;
    m_dy=dy;
  }
  virtual void Draw();
  virtual void Box(float &x1, float &y1, float &x2, float &y2);
};

class CEllipse:public CShape
{
  float m_rx;
  float m_ry;
public:
  CEllipse(float x, float y, float rx, float ry) : CShape(x,y)
  {
    m_rx=rx;
    m_ry=ry;
  }
  virtual void Draw();
  virtual void Box(float &x1, float &y1, float &x2, float &y2);
};

void CShape::Box(float &x1, float &y1, float &x2, float &y2)
{
   x1=m_x; y1=m_y; x2=m_x; y2=m_y;
}

void CLine::Draw()
{
  printf("Rysuję odcinek od punktu (%f, %f) do punktu (%f, %f)\n",
         m_x,m_y,m_x2,m_y2);
}
```

```
void CLine::Box(float &x1, float &y1, float &x2, float &y2)
{
   x1=m_x; y1=m_y; x2=m_x2; y2=m_y2;
}

void CRectangle::Draw()
{
  printf("Rysuję prostokąt o środku (%f, %f) i wielkości %f, %f\n",
         m_x,m_y,m_dx,m_dy);
}

void CRectangle::Box(float &x1, float &y1, float &x2, float &y2)
{
   x1=m_x-m_dx/2; y1=m_y-m_dy/2; x2=m_x+m_dx/2; y2=m_y+m_dy/2;
}

void CEllipse::Draw()
{
  printf("Rysuję elipsę o środku (%f, %f) i promieniach (%f, %f)\n",
         m_x,m_y,m_rx,m_ry);
}

void CEllipse::Box(float &x1, float &y1, float &x2, float &y2)
{
   x1=m_x-m_rx; y1=m_y-m_ry; x2=m_x+m_rx; y2=m_y+m_ry;
}

#pragma argsused
int main(int argc, char **argv)
{
  CShape *figury[10];
  int i;
  float xmin,ymin,xmax,ymax;
  float x1,y1,x2,y2;

  figury[0]=new CLine(10,10,30,40);
  figury[1]=new CRectangle(50,50,25,35);
  figury[2]=new CEllipse(40,40,10,20);

  for (i=0; i<3; i++)
    figury[i]->Draw();

  figury[0]->Box(xmin,ymin,xmax,ymax);
  for (i=1; i<3; i++)
  {
    figury[i]->Box(x1,y1,x2,y2);
    if (x1<xmin) xmin=x1;
    if (x2>xmax) xmax=x2;
    if (y1<ymin) ymin=y1;
    if (y2>ymax) ymax=y2;
  }

  printf("Wszystkie figury mieszczą się w obszarze (%f,%f)-(%f,%f)\n",
         xmin,ymin,xmax,ymax);

  for (i=0; i<3; i++)
```

```
    delete figury[i];

  return 0;
}
```

17 Programy wielomodułowe

Programu obiektowe to zazwyczaj większe programy. Pisanie całego programu w jednym pliku teoretycznie jest możliwe a w praktyce niespotykane. Szczególnie gdy większy projekt jest dzielony pomiędzy kilku programistów pracujących zespołowo. Dzieląc program na mniejsze fragmenty staramy się dzielić go na klasy, a każdą klasę umieszczamy w odrębnym module[5]. Klasę `CComplex` będziemy deklarować w module `complex.cpp`, a deklaracje związane z tą klasą dostępne w innych modułach umieścimy w pliku nagłówkowym `complex.h`.

Plik nagłówkowy `complex.h` będzie deklarował klasę `CComplex`.

```
//deklaracja klasy CComplex i definicje metod otwartych tej
//klasy.
class CComplex
{
  double m_re;
  double m_im;
public:
  CComplex(double re, double im) { m_re=re; m_im=im; }
  void set(double re, double im) { m_re=re; m_im=im; }
  void  print();
  double mod();
};
```

Plik definiujący metody klasy `CComplex` o nazwie `complex.h` będzie wyglądał w sposób następujący.

```
#include <stdio.h>
#include <math.h>
#include "complex.h"

void CComplex::print()
{
  printf("%f+i*%f\n", m_re,m_im);
}

double CComplex::mod()
{
  return sqrt(m_re*m_re+m_im*m_im);
}
```

Ostatni plik to przykładowy progarm korzystający z klasy `CComplex`:

```
#include <stdio.h>
#include "complex.h"
```

[5] Jeśli są to małe klasy to oczywiście nie musimy dzielić na tak małe pliki. Staramy się łączyć klasy w jednym module według ich wzajemnych powiązań.

```
int main(int /*argc*/, char **/*argv*/)
{
  CComplex a(2.0,1.0);

  a.print();
  printf("modul a=%f\n", a.mod());
  return 0;
}
```

Dla takiego wielomodułowego programu trzeba przygotować odpowiedni plik z komendami `make`-a (np. o nazwie `zespo.mak`).

```
CC=c:\borland\bcc55\bin\bcc32
LINK=c:\borland\bcc55\bin\ilink32
LIB=c:\borland\bcc55\lib
INC=c:\borland\bcc55\include
STDLIB=cw32.lib import32.lib
CCOPT=-5 -v -tWC
LINKOPT=-v

.cpp.obj:
  $(CC) -c $(CCOPT) -I$(INC) $<

zespo.exe: zespo.obj complex.obj
  $(LINK) -c $(LINKOPT) -L$(LIB) c0x32.obj zespo.obj complex.obj, \
                                          zespo.exe,zespo.map,$(STDLIB)

zespo.obj: zespo.cpp complex.h
complex.obj: complex.cpp complex.h
```

Po napisaniu takiego pliku przeprowadzamy kompilację poleceniem `make -fzespo.mak`. Proszę zwrócić owagę, że w skład programu wchodzą trzy skompilowane moduły: kod uruchomieniowy `c0x32.obj` oraz dwa własne moduły `zespo.obj` i `complex.obj`. Zmieniają się również zależności. Plik wykonywalny zależy od naszych modułów: `zespo.obj` i `complex.obj`. Natomiast moduł `zespo.obj` trzeba przekompilować gdy zmieni się `zespo.cpp` lub `complex.h`, a moduł `complex.obj` trzeba skompilować gdy zmieni się `complex.cpp` lub `complex.h`.

18 Przykłady klas